

PART 1



Introducing .NET



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/htm. 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 `<input type="checkbox"/>` 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.

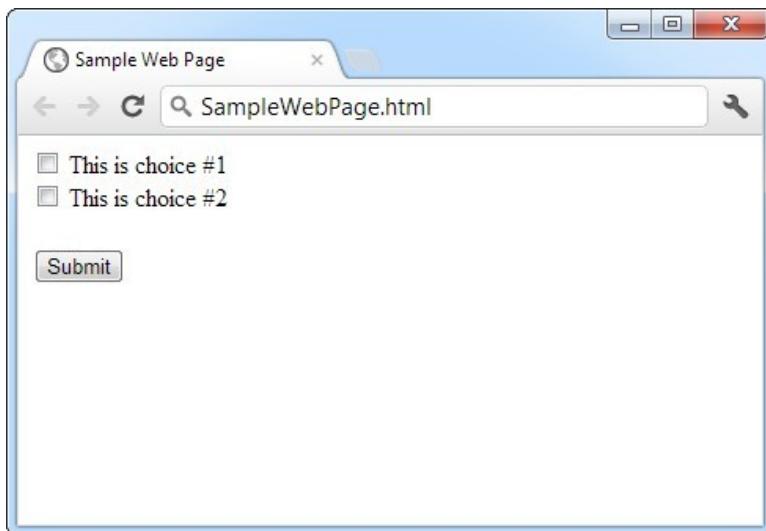


Figure 1-2. An HTML form

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 by all browsers (and even the browsers that support them aren't always consistent). However, ASP.NET will use the 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.

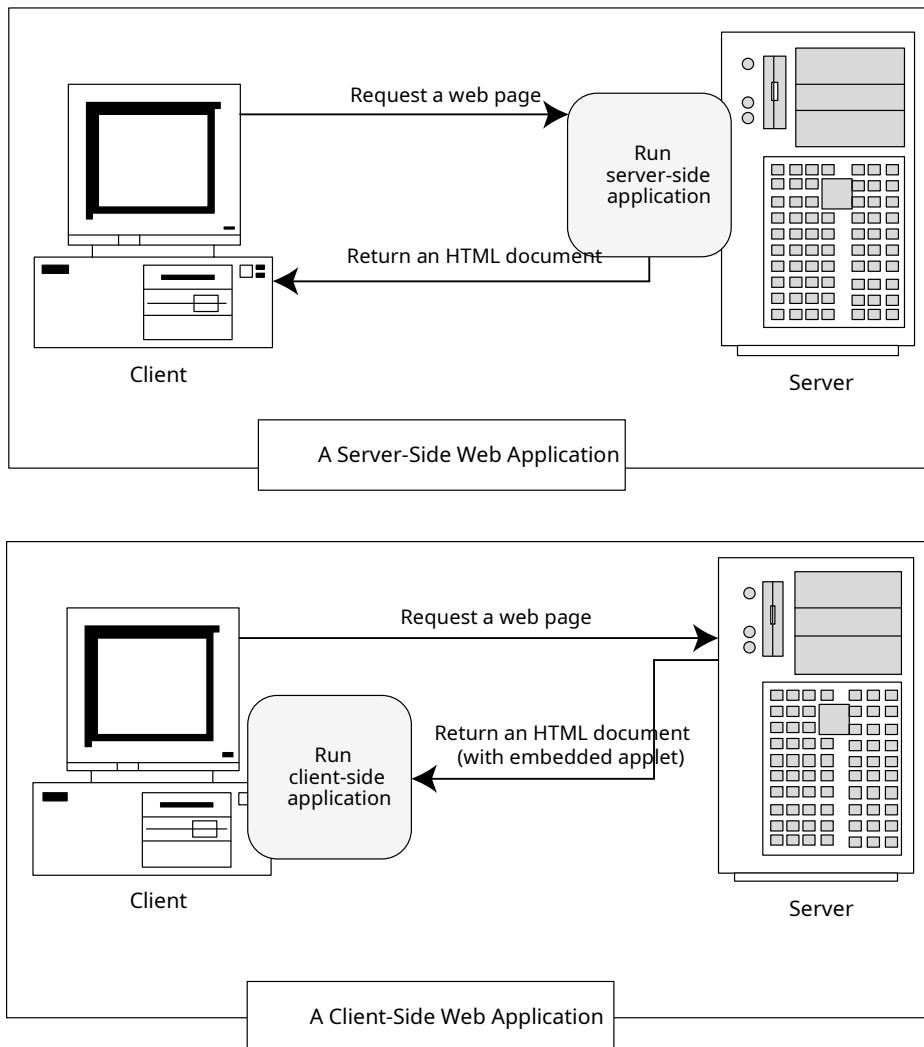


Figure 1-3. Server-side and client-side web applications

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 real-time browser-based game or the next Google Maps. If this is what you want, it's largely up to you to write huge amounts of complex JavaScript that you need to your ASP.NET web forms. However, if you'd prefer to build 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.

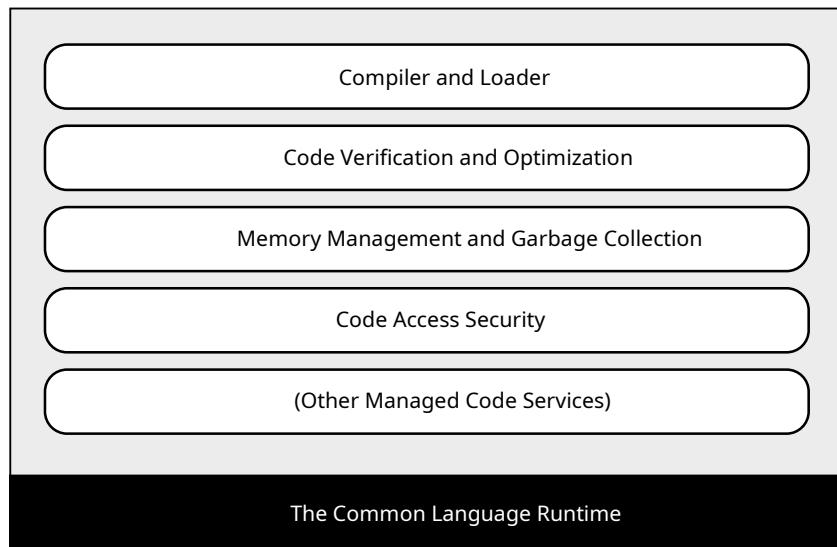
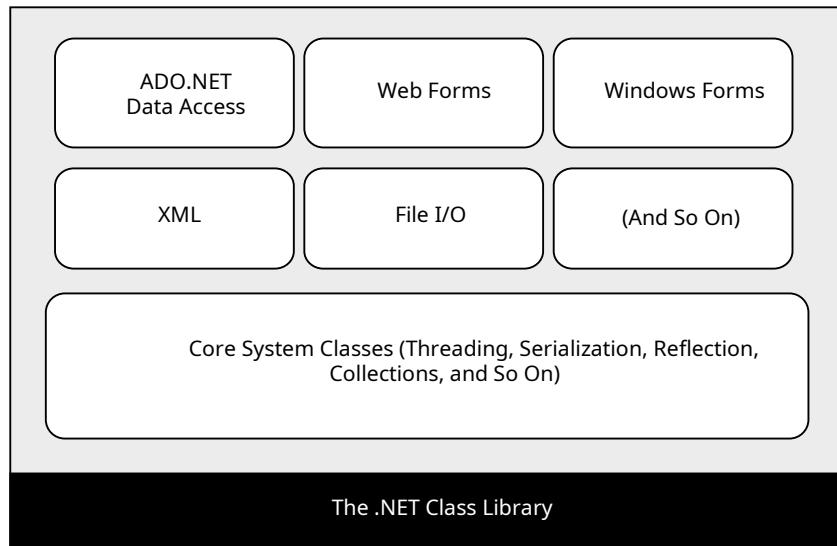


Figure 1-4. The .NET Framework

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 the framework 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.

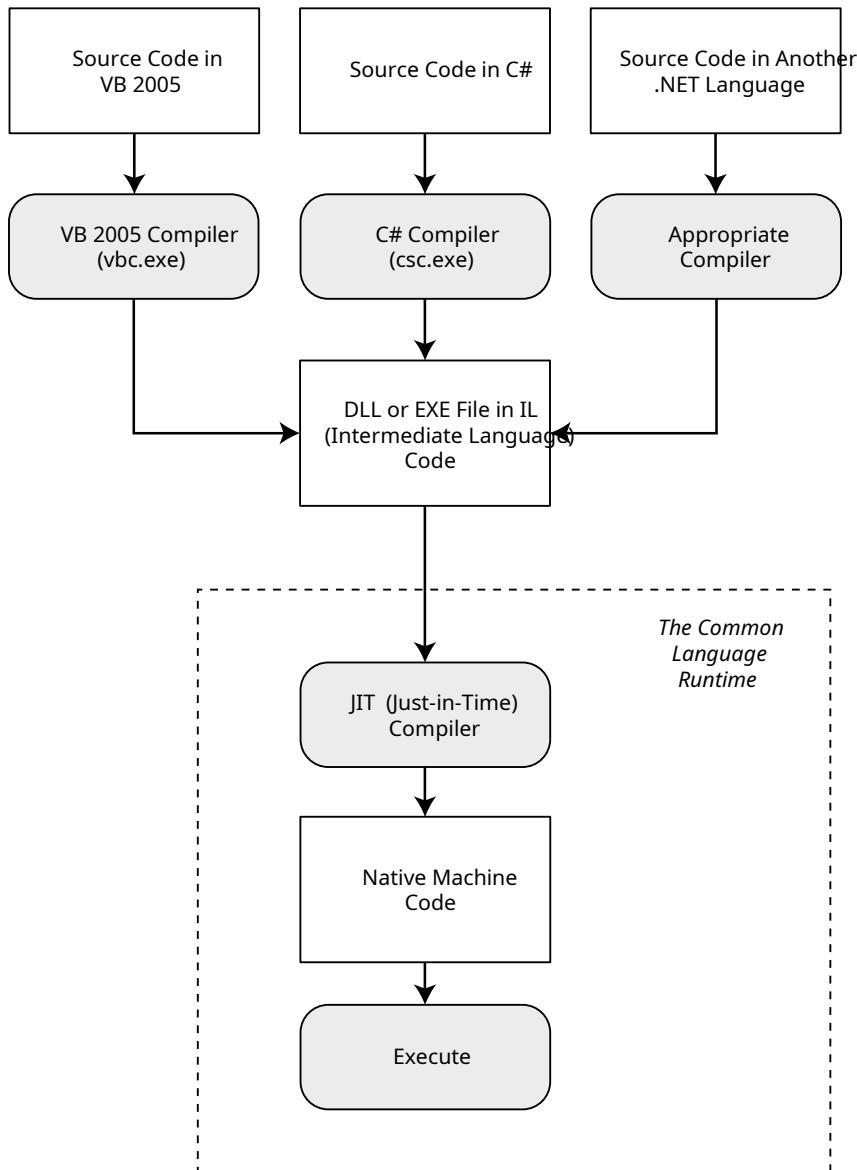


Figure 1-5. Language compilation in .NET

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 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 surprisingly 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 <http://go.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 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.



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 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 you want to sell to other companies), coding standards are particularly important. But even if you aren't, consistent coding is a good habit that will make it easier for you to understand the code you've written now (and even years!) later. You can find a good summary of best practices in the "IDesign C# Coding Standard" written 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~~* 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 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, 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 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, a Microsoft-designed ECMA standard that sets the ground rules that all .NET languages must follow when working 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 beginning of a variable name to indicate its data type. In .NET, this practice is discouraged, because data types can now 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` was 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.

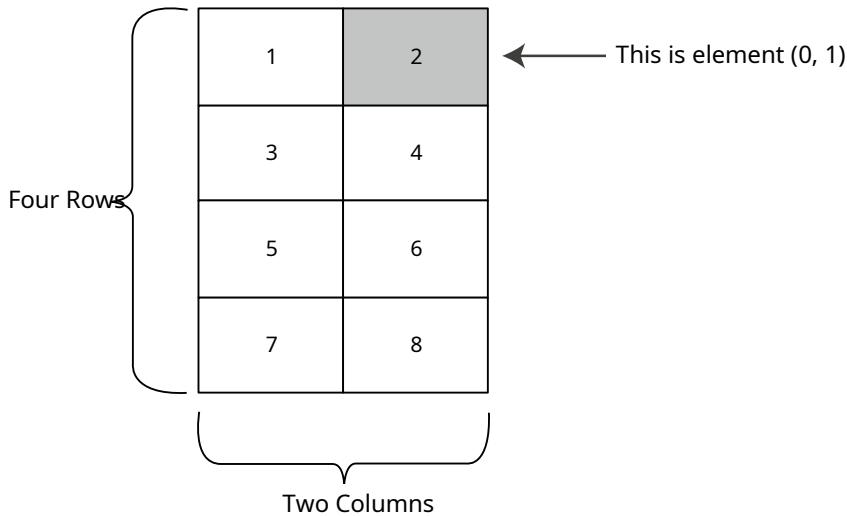


Figure 2-1. A sample array of integers

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 arrays. 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, such as 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 the previous section, enumerations are sets of constant values. However, programmers often talk about the process of enumerating over a collection, which means to loop, or *iterate*, over a collection. For example, it's common to talk about enumerating over the characters of a string (which means looping through the string and examining each character in a separate loop).

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 applications, unless you're designing your own components. However, the concept of enumerated values is very important, because the .NET class library uses it extensively. For example, you set colors, border styles 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 5.0, 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 `try...catch`, because it's inefficient. `try...catch` catches the problem (preventing a data error), but it throws an exception, which you need to handle by using error-handling code 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, `try...catch` 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 used to access a piece of data that's associated with an object. You'll learn more about methods and properties in the 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 <i>pre</i> 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 <i>a</i> characters in a string into <i>b</i> 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 <i>pre</i> 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 `+` 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 <i>Coordinated Universal Time</i> (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 <i>TimeSpan</i> . 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 <i>TimeSpan</i> 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 <i>TimeSpan</i> 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 <i>TimeSpan</i> . For example, you can use <i>TimeSpan.FromHours(24)</i> to create a <i>TimeSpan</i> object exactly 24 hours long.
ToString()	Returns a string representation of the current <i>TimeSpan</i> 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
<code>Length</code>	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.
<code>GetLowerBound()</code> and <code>GetUpperBound()</code>	Determines the dimensions of an array. As with just about everything in .NET, you start counting at zero (which represents the first dimension).
<code>Clear()</code>	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).
<code>IndexOf()</code> and <code>LastIndexOf()</code>	Searches a one-dimensional array for a specified value and returns the index number. You cannot use this with multidimensional arrays.
<code>Sort()</code>	Sorts a one-dimensional array made up of comparable data such as strings or numbers.
<code>Reverse()</code>	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
<code>==</code>	Equal to.
<code>!=</code>	Not equal to.
<code><</code>	Less than.
<code>></code>	Greater than.
<code><=</code>	Less than or equal to.
<code>>=</code>	Greater than or equal to.
<code>&&</code>	Logical and (evaluates to true only if both expressions are true). If the first expression is false, the second expression is not evaluated.
<code> </code>	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 braces. 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 line, 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~~~~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~~ 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.WriteLine(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 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.WriteLine(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 `return` 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 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 number of parameters while centralizing functionality under common names. Even the methods you've seen so far (such as String methods for padding or replacing text) have multiple versions that provide similar features with different parameter lists.

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 therefore 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.



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*.

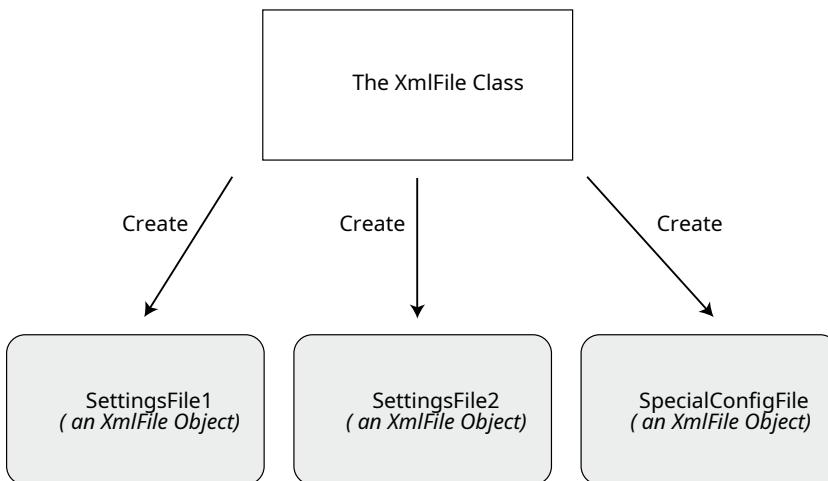


Figure 3-1. Classes are used to create objects

Note At its simplest, *object-oriented programming* is the idea that your code should be organized into 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) <i>and</i> 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 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 like. 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 .NET runtime uses a long-running service (called *garbage collection*) that periodically scans for released objects and recycles 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` parameter. 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 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 ChangeDetected 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 to 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 `-` 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 similar¹). The completed page is then sent to the user. The end result is that you work with objects, instead of dealing directly with raw HTML code.

¹Actually, 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~~ **override**, 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.

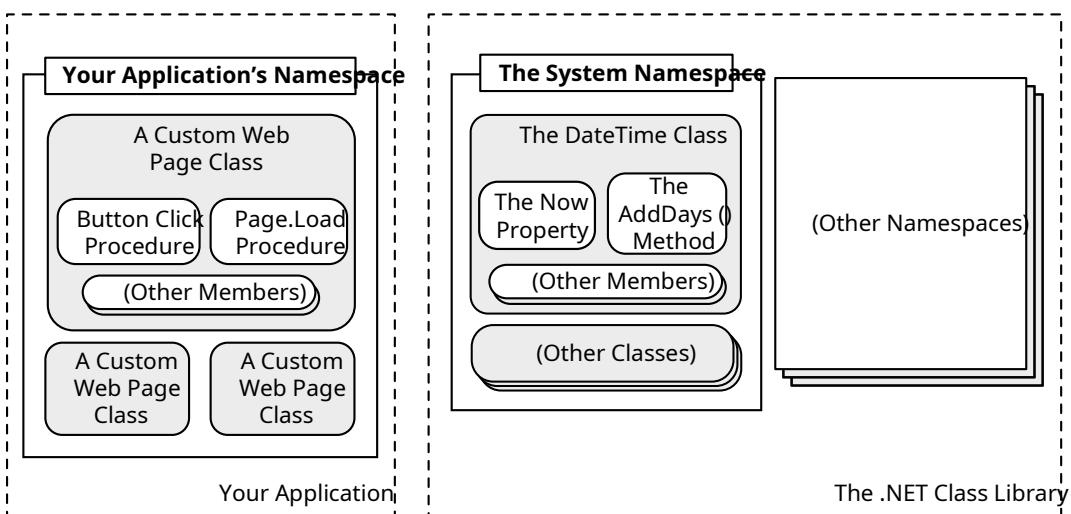


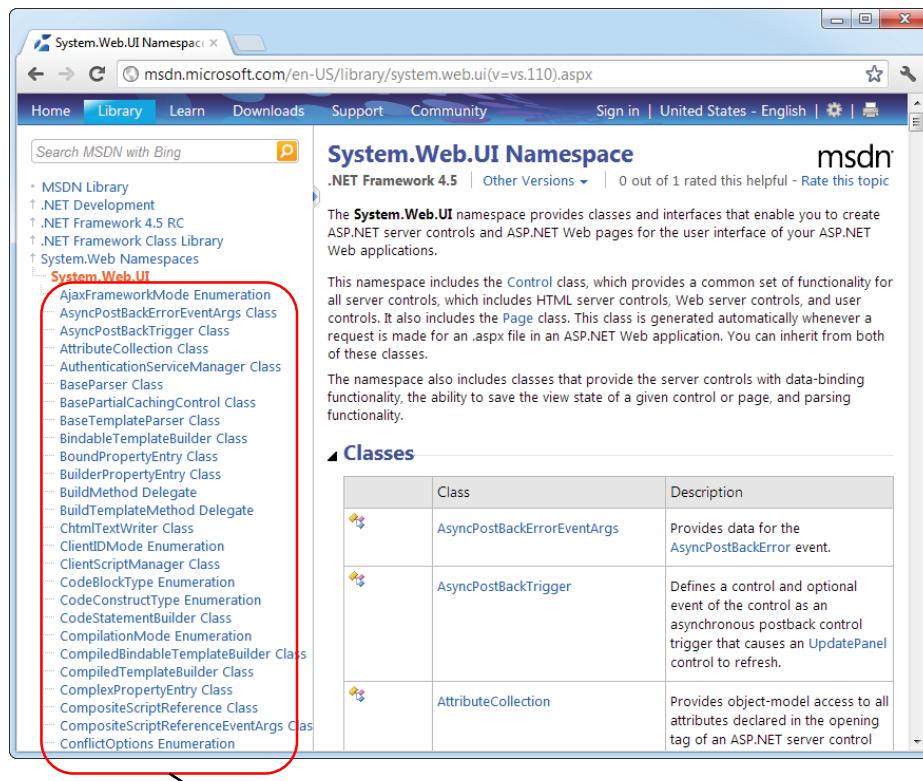
Figure 3-3. A look at two namespaces

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.



The System.Web.UI namespace provides classes and interfaces that enable you to create ASP.NET server controls and ASP.NET Web pages for the user interface of your ASP.NET Web applications.

This namespace includes the `Control` class, which provides a common set of functionality for all server controls, which includes HTML server controls, Web server controls, and user controls. It also includes the `Page` class. This class is generated automatically whenever a request is made for an .aspx file in an ASP.NET Web application. You can inherit from both of these classes.

The namespace also includes classes that provide the server controls with data-binding functionality, the ability to save the view state of a given control or page, and parsing functionality.

Classes

	Class	Description
	<code>AsyncPostBackEventArgs</code>	Provides data for the <code>AsyncPostBackError</code> event.
	<code>AsyncPostBackTrigger</code>	Defines a control and optional event of the control as an asynchronous postback control trigger that causes an <code>UpdatePanel</code> control to refresh.
	<code>AttributeCollection</code>	Provides object-model access to all attributes declared in the opening tag of an ASP.NET server control

The long list of types in this namespace (it continues if you scroll down the page)

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.

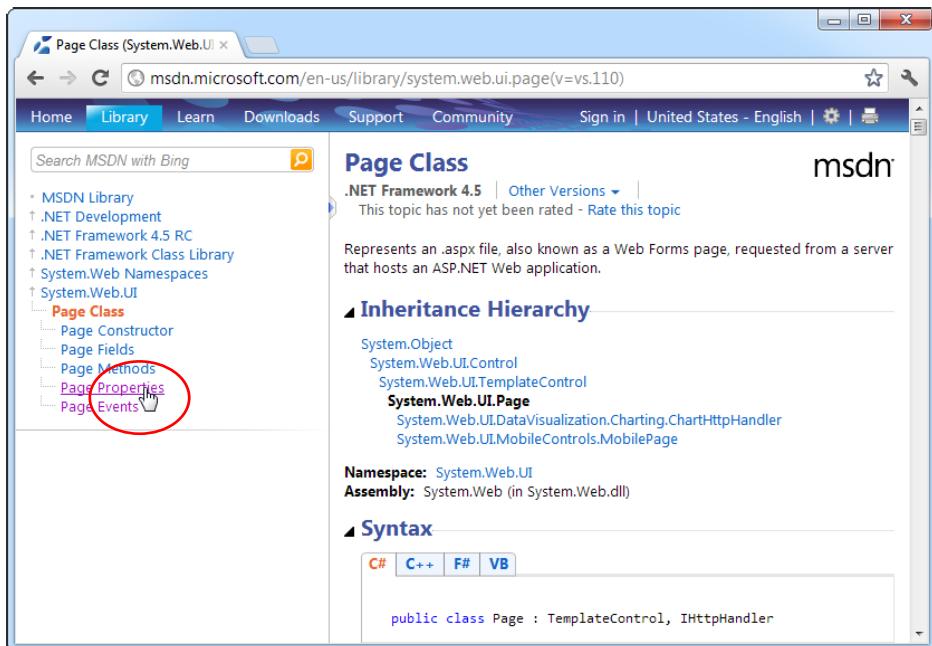


Figure 3-5. Exploring the members of a specific class

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 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-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 this reason, neither of these statements is allowed:

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

Furthermore, this 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 run as 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 mscorelib.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 (the TotalPrice property and the three-argument constructor). They're just left out of the listing to focus on 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 by adding 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 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, 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 do an application that uses the `Product` object. That application is actually able to handle an instance of any 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 have 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 (it isn't terribly common). However, the real answer is that generics weren't implemented in .NET until version 2.0, and 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 have yet to learn. If you want to continue your exploration of C# and become a language guru, you can visit microsoft's C# developer Center <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).



Web Form Fundamentals

In this chapter, you'll learn some of the core topics that every ASP.NET developer must master. You'll begin by taking a closer look at the ASP.NET application model, and considering what files and folders belong in a web application. Then you'll take a closer look at *server controls*, the basic building block of every web form. Using server controls, you'll create your first ASP.NET application, by taking a static HTML page and transforming it into a simple, single-page currency converter.

After you have the currency converter example under your belt, you can explore the basics of ASP.NET's web form model in more detail. You'll pick up some handy skills along the way, such as the ability to create controls on the fly, navigate from one page to another, and handle special characters in HTML. Finally, you'll consider the ASP.NET configuration model, which lets you tweak the settings that govern the behavior of your web application.

Understanding the Anatomy of an ASP.NET Application

It's sometimes difficult to define exactly what a web application is. Unlike a desktop program or smartphone app, ASP.NET websites almost always include multiple web pages. This division means a user can enter an ASP.NET "application" at different points, and follow a link from one page to another part of the website or to another web server. So, does it make sense to treat an entire website as though it were a single application?

In ASP.NET, the answer is yes. Every ASP.NET application shares a common set of resources and configuration settings. Web pages from other ASP.NET applications don't share these resources, even if they're on the same web server. Technically speaking, every ASP.NET application is executed inside a separate *application domain*. Application domains are isolated areas in memory, and they ensure that even if one web application causes a fatal error, it's unlikely to affect any other application that is currently running on the same computer (in this case, that's the web server). Similarly, application domains restrict a web page in one application from accessing the in-memory information of another application. Each web application is maintained separately and has its own set of cached, application, and session data.

The standard definition of an ASP.NET application describes it as a combination of files, pages, handlers, modules, and executable code that can be invoked from a virtual directory (and, optionally, its subdirectories) on a web server. In other words, the virtual directory is the basic grouping structure that delimits an application. Figure 5-1 shows a web server that hosts four separate web applications.

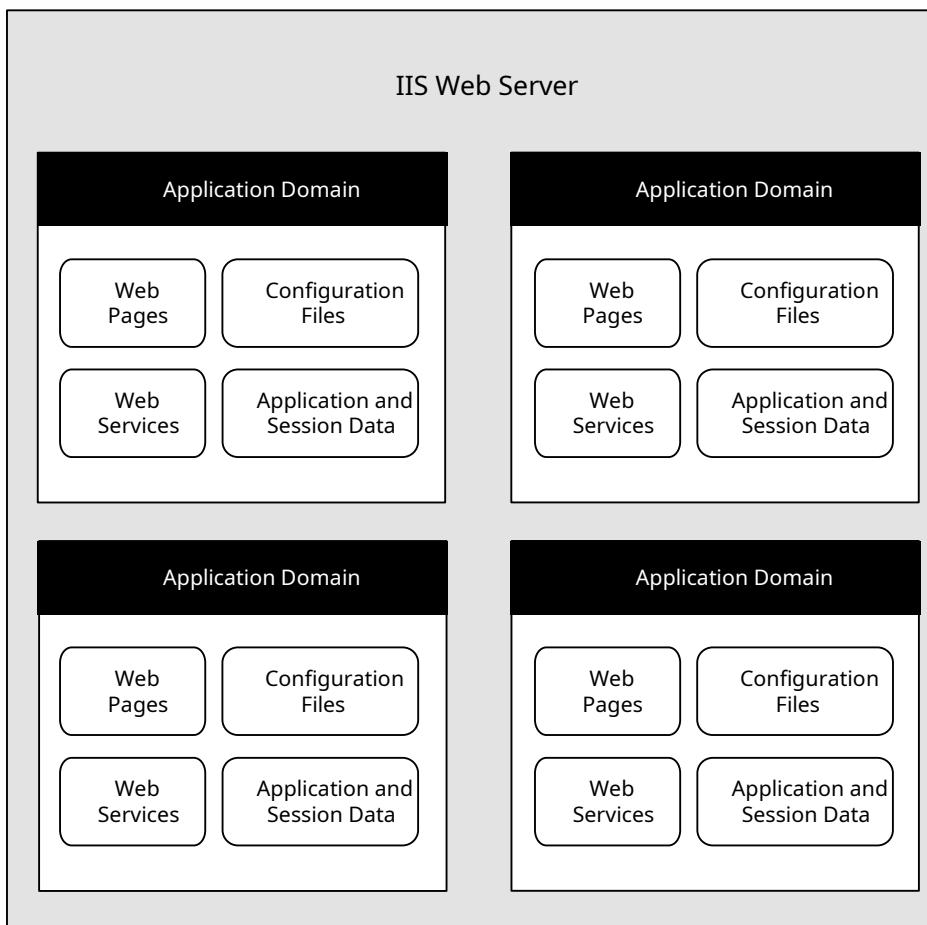


Figure 5-1. ASP.NET applications

Note A virtual directory is a directory that's exposed to other computers on a web server. As you'll discover in Chapter 26, you deploy your perfected ASP.NET web application by copying it to a virtual directory.

ASP.NET File Types

ASP.NET applications can include many types of files. Table 5-1 introduces the essential ingredients.

Table 5-1. ASP.NET File Types

File Name	Description
Ends with .aspx	These are ASP.NET web pages. They contain the user interface and, optionally, the underlying application code. Users request or navigate directly to one of these pages to start your web application.
Ends with .ascx	These are ASP.NET user controls. User controls are similar to web pages, except that the user can't access these files directly. Instead, they must be hosted inside an ASP.NET web page. User controls allow you to develop a small piece of user interface and reuse it in as many web forms as you want without repetitive code. You'll learn about user controls in Chapter 11.
web.config	This is the configuration file for your ASP.NET application. It includes settings for customizing security, state management, memory management, and much more. You'll get an introduction to the web.config file in this chapter, and you'll explore its settings throughout this book.
global.asax	This is the global application file. You can use this file to define global variables (variables that can be accessed from any web page in the web application) and react to global events (such as when a web application first starts). You'll learn about it later in this chapter.
Ends with .cs	These are code-behind files that contain C# code. They allow you to separate the application logic from the user interface of a web page. We'll introduce the code-behind model in this chapter and use it extensively in this book.

In addition, your web application can contain other resources that aren't special ASP.NET files. For example, your virtual directory can hold image files, HTML files, or CSS style sheets. These resources might be used in your ASP.NET web pages, or they might be used independently. A website can even combine static HTML pages with dynamic ASP.NET pages.

Note To access an AsP.nET website, a visitor browses to an .aspx page. The other file types that are listed in Table 5-1 cannot be accessed directly. If a visitor attempts to request one of these files, the web server will deny the request (to ensure good security). You'll learn more about how web servers deal with AsP.nET file types in Chapter 26, when you learn to deploy a completed web application.

ASP.NET Web Folders

Every web application starts with a single location, called the *root folder*. However, in a large, well-planned web application, you'll usually create additional folders inside the website's root folder. For example, you'll probably want to store images in a subfolder while you place web pages in the root folder. Or you might want to put public ASP.NET pages in one subfolder and restricted ones in another so you can apply different security settings based on the folder. (See Chapter 19 for more about how to create authorization rules like this.)

Tip To create a subfolder in Visual studio, right-click your website in the solution Explorer and choose new Folder.

Along with the custom folders you create, ASP.NET also uses a few specialized folders, which it recognizes by name (see Table 5-2). Keep in mind that you won't see all these folders in a typical application. In fact, in a brand

Table 5-2. ASP.NET Folders

Directory	Description
App_Browsers	Contains .browser files that ASP.NET uses to identify the browsers that are using your application and determine their capabilities. Usually, browser information is standardized across the entire web server, and you don't need to use this folder. For more information about ASP.NET's browser support—which is an advanced feature that most ordinary web developers can safely ignore—refer to <i>Pro ASP.NET 4.5 in C#</i> (Apress, 2012).
App_Code	Contains source code files that are dynamically compiled for use in your application.
App_GlobalResources	Stores global resources that are accessible to every page in the web application. This directory is used in localization scenarios, when you need to have a website in more than one language. Localization isn't covered in this book, and you can refer to <i>Pro ASP.NET 4.5 in C#</i> for more information.
App_LocalResources	Serves the same purpose as App_GlobalResources, except these resources are accessible to a specific page only.
App_WebReferences	Stores references to web services, which are remote code routines that a web application can call over a network or the Internet.
App_Data	Stores data, including SQL Server Express database files (as you'll see in Chapter 14). Of course, you're free to store data files in other directories.
App_Themes	Stores the themes that are used to standardize and reuse formatting in your web application. You'll learn about themes in Chapter 12.
Bin	Contains all the compiled .NET components (DLLs) that the ASP.NET web application uses. For example, if you develop a custom component for accessing a database (see Chapter 22), you'll place the component here. ASP.NET will automatically detect the assembly, and any page in the web application will be able to use it.

new blank website—such as the one you learned to create in Chapter 4—you won't have any subfolders. Instead, Visual Studio will create them when you need them.

Introducing Server Controls

In old-style web development, programmers had to master the quirks and details of HTML before they could design a dynamic web page. Pages had to be carefully tailored to a specific task, and the only way to add new content to a page was to generate raw HTML tags.

ASP.NET solves this problem with a higher-level model of *server controls*. These controls are created and configured as *objects*. They run on the web server and automatically provide their own HTML output in a way

that's conceptually similar to the simple garbage can example you saw in Chapter 2. Even better, server controls behave like their Windows counterparts by maintaining state and raising events that you can react to in code.

In Chapter 3, you built an exceedingly simple web page that incorporated a few controls you dragged in from the Visual Studio Toolbox. But before you create a more complex page, it's worth taking a step back to look at the big picture. ASP.NET actually provides two sets of server-side controls that you can incorporate into your web forms. These two types of controls play subtly different roles:

HTML server controls: These are server-based equivalents for standard HTML elements. These controls are ideal if you're a seasoned web programmer who prefers to work with familiar HTML tags (at least at first). They are also useful when migrating ordinary HTML pages or classic ASP pages to ASP.NET, because they require the fewest changes.

Web controls: These are similar to the HTML server controls, but they provide a richer object model with a variety of properties for style and formatting details. They also provide more events and more closely resemble the controls used for Windows development. Web controls also feature some user interface elements that have no direct HTML equivalent, such as the GridView, Calendar, and validation controls.

You'll learn about web controls in the next chapter. In this chapter, you'll take a detailed look at HTML server controls.

Note Even if you plan to use web controls exclusively, it's worth reading through this section to master the basics of HTML server controls. Along the way, you'll get an introduction to a few ASP.NET essentials that apply to all kinds of server controls, including view state, postbacks, and event handling.

HTML Server Controls

HTML server controls provide an object interface for standard HTML elements. They provide three key features:

They generate their own interface: You set properties in code, and the underlying HTML tag is created automatically when the page is rendered and sent to the client.

They retain their state: Because the Web is stateless, ordinary web pages need to do a lot of work to store information between requests. HTML server controls handle this task automatically. For example, if the user selects an item in a list box, that item remains selected the next time the page is generated. Or if your code changes the text in a button, the new text sticks the next time the page is posted back to the web server.

They fire server-side events: For example, buttons fire an event when clicked, text boxes fire an event when the text they contain is modified, and so on. Your code can respond to these events, just like ordinary controls in a Windows application. If a given event doesn't occur, the event handler won't be executed.

HTML server controls are ideal when you're performing a quick translation to add server-side code to an existing HTML page. That's the task you'll tackle in the next section, with a simple one-page web application.

Converting an HTML Page to an ASP.NET Page

Figure 5-2 shows a currency converter web page. It allows the user to convert US dollars to euros—or at least it would, if it had the code it needed to do the job. Right now, it's just a plain, inert HTML page. Nothing happens when the button is clicked.

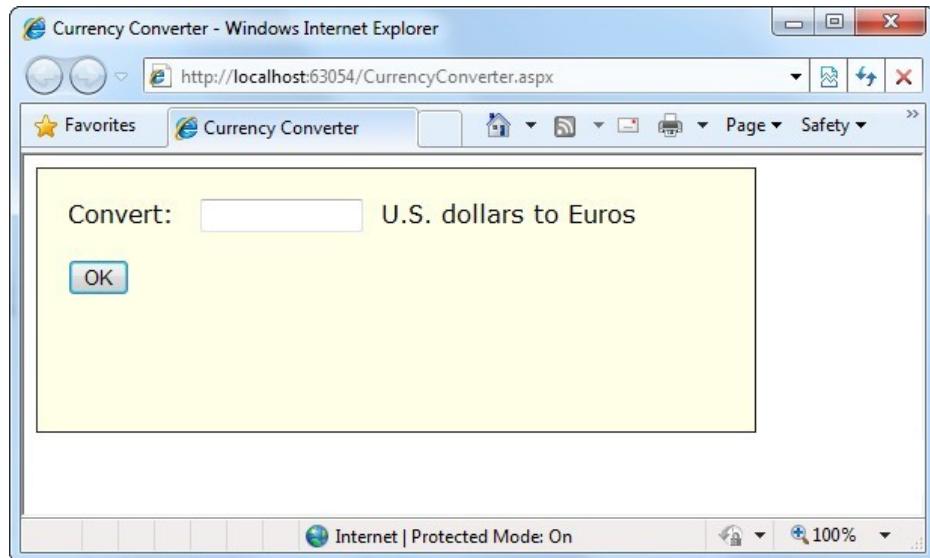


Figure 5-2. A simple currency converter

The following listing shows the markup for this page. To make it as clear as possible, this listing omits the style attribute of the `<div>` element used for the border. This page has two `<input>` elements: one for the text box and one for the submit button. These elements are enclosed in a `<form>` tag, so they can submit information to the server when the button is clicked. The rest of the page consists of static text. The ` ` character entity is used to add an extra space to separate the controls. A doctype at the top of the page declares that it's written according to the modern HTML5 standard.

```
<!DOCTYPE html>
<html>
<head>
<title> Currency Converter</title>
</head>
<body>
<form method = "post">
<div>
    Convert:&nbsp;
    <input type = "text" />
    &nbsp;U.S. dollars to Euros.
    <br /> <br />
    <input type = "submit" value = "OK" />
</div>
</form>
</body>
</html>
```

Note In HTMl, many input controls are represented by the < input > element. You set the type attribute to indicate the type of control you want. The < input type = "text" > tag is a text box, while < input type = "submit" > creates a submit button for sending the web page back to the web server. This is quite a bit different from the way you'll see in Chapter 6, which use a different element for each type of control.

As it stands, this page looks nice but provides no functionality. It consists entirely of the user interface (HTML elements) and contains no code. It's an ordinary HTML page—not a web form.

The easiest way to convert the currency converter to ASP.NET is to start by generating a new web form in Visual Studio. To do this, follow these steps:

1. Right-click your website in the Solution Explorer and choose **Add New Item**.
2. In the Add New Item dialog box, choose Web Form (which should be first in the list).
3. Type a name for the new page (such as CurrencyConverter.aspx).
4. Make sure the Place Code in Separate File option is selected (it's in the bottom-right corner of the window).
5. Click Add to create the page.

In the new web form, delete everything that's currently in the .aspx file, except the page directive. The *page directive* gives ASP.NET basic information about how to compile the page. It indicates the language you're using for your code and the way you connect your event handlers. If you're using the code-behind approach, which is recommended, the page directive also indicates where the code file is located and the name of your custom page class.

Finally, copy all the content from the original HTML page, and paste it into the new page, right after the page directive. Here's the resulting web form, with the page directive (in bold) followed by the HTML content that's copied from the original page:

```
<%@ Page Language = "C#" AutoEventWireup = "true"
  CodeFile = "CurrencyConverter.aspx.cs" Inherits = "CurrencyConverter" %>
<!DOCTYPE html>
<html>
<head>
  <title> Currency Converter</title>
</head>
<body>
  <form method = "post">
    <div>
      Convert: &nbsp;
      <input type = "text" />
      &nbsp; U.S. dollars to Euros.
      <br /> <br />
      <input type = "submit" value = "OK" />
    </div>
  </form>
</body>
</html>
```

Now you need to add the attribute runat = "server" to each tag that you want to transform into a server control. You should also add an ID attribute to each control that you need to interact with in code. The ID attribute assigns the unique name that you'll use to refer to the control in code.

In the currency converter application, it makes sense to change the input text box and the submit button into HTML server controls. In addition, the `<form>` element must be processed as a server control to allow ASP.NET to access the controls it contains. Here's the complete, correctly modified page:

```
<%@ Page Language = "C#" AutoEventWireup = "true"
  CodeFile = "CurrencyConverter.aspx.cs" Inherits = "CurrencyConverter" %>

<!DOCTYPE html>
<html>
  <head>
    <title> Currency Converter</title>
  </head>
  <body>
    <form runat = "server">
      <div>
        Convert: &nbsp;
        <input type = "text" ID = "US" runat = "server" />
        &nbsp; U.S. dollars to Euros.
        <br /> <br />
        <input type = "submit" value = "OK" ID = "Convert" runat = "server" />
      </div>
    </form>
  </body>
</html>
```

Note most ASP.NET controls must be placed inside the `<form>` section in the page. The `<form>` element is part of the standard for HTML forms, and it allows the browser to send information to the web server.

The web page still won't do anything when you run it, because you haven't written any code. However, now that you've converted the static HTML elements to HTML server controls, you're ready to work with them.

View State

To try this page, launch it in Visual Studio by clicking the Play button in the standard Visual Studio toolbar (or just press F5). Remember, the first time you run your web application, you'll be prompted to let Visual Studio modify your `web.config` file to allow debugging. Click OK to accept its recommendation and launch your web page in the browser. Then, right-click the page and choose View Source in your browser to look at the HTML that ASP.NET sent your way.

The first thing you'll notice is that the HTML that was sent to the browser is slightly different from the information in the `.aspx` file. First, the `runat = "server"` attributes are stripped out (because they have no meaning to the client browser, which can't interpret them). Second, and more important, an additional hidden field has been added to the form. Here's what you'll see (in a slightly simplified form):

```
<!DOCTYPE html>
<html>
  <head>
    <title> Currency Converter</title>
  </head>
```

```

<body>
<form ID = "form1" method = "post" action = "CurrencyConverter.aspx">
<div class = "aspNetHidden">
    <input type = "hidden" name = "__VIEWSTATE" ID = "__VIEWSTATE"
        value = "dDw3NDg2NTI5MDg7Oz4..." />
</div>
<div class = "aspNetHidden">
    <input type = "hidden" name = "__EVENTVALIDATION" ID = "__EVENTVALIDATION"
        value = "/wEWAwLr3rrOBgLr797..." />
</div>
<div>
    Convert: &nbsp;
    <input type = "text" ID = "US" name = "US" />
    &nbsp; U.S. dollars to Euros.
    <br /> <br />
    <input type = "submit" value = "OK" ID = "Convert" name = "Convert" />
</div>
</form>
</body>
</html>

```

Keen eyes will notice that ASP.NET has added two new hidden `<input>` elements, each of which is nested in a separate `<div>` element container. The second hidden field is used to stop certain types of web page tampering. It works behind the scenes. More interesting is the first hidden field, which stores information, in a compressed format, about the state of every control in the page:

```

<input type = "hidden" name = "__VIEWSTATE" ID = "__VIEWSTATE"
value = "dDw3NDg2NTI5MDg7Oz4..." />

```

This information is called the *view state* of the page.

Because the view state information is stored in your page, your code can change control properties at any time, and these changes will automatically “stick.” For example, if you change the background color of a box, that box will keep its new color, no matter how many times the page is sent back and forth between the browser and the web server. This is a key part of the web forms programming model. Thanks to view state, you can often forget about the stateless nature of the Internet and treat your page like a continuously running application.

Even though the currency converter program doesn’t yet include any code, you’ll already notice one change. If you enter information in the text box and click the submit button to post the page, the refreshed page will still contain the value you entered in the text box. (In the original example that uses ordinary HTML elements, the value is cleared every time the page is submitted.) This change occurs because ASP.NET controls automatically retain their state.

The HTML Control Classes

Before you can continue any further with the currency converter, you need to know about the control objects you’ve created. All the HTML server controls are defined in the `System.Web.UI.HtmlControls` namespace. Each kind of control has a separate class. Table 5-3 describes the basic HTML server controls and shows you the related HTML element.

Table 5-3. The HTML Server Control Classes

Class Name	HTML Element	Description
HtmlForm	<form>	Wraps all the controls on a web page. All ASP.NET server controls must be placed inside an HtmlForm control so that they can send their data to the server when the page is submitted. Visual Studio adds the <form> element to all new pages. When designing a web page, you need to ensure that every other control you add is placed inside the <form> section.
HtmlAnchor	<a>	A hyperlink that the user clicks to jump to another page.
HtmlImage		A link that points to an image, which will be inserted into the web page at the current location.
HtmlTable, HtmlTableRow, and HtmlTableCell	<table>, <tr>, <th>, and <td>	A table that displays multiple rows and columns of static text.
HtmlInputButton, HtmlInputSubmit, and HtmlInputReset	<input type="button">, <input type="submit">, and <input type="reset">	A button that the user clicks to perform an action (HtmlInputButton), submit the page (HtmlInputSubmit), or clear all the user-supplied values in all the controls (HtmlInputReset).
HtmlButton	<button>	A button that the user clicks to perform an action. This is not supported by all browsers, so HtmlInputButton is usually used instead. The key difference is that the HtmlButton is a container element. As a result, you can insert just about anything inside it, including text and pictures. The HtmlInputButton, on the other hand, is strictly text-only.
HtmlInputCheckBox	<input type="checkbox">	A check box that the user can select or clear. Doesn't include any text of its own.
HtmlInputRadioButton	<input type="radio">	A radio button that can be selected in a group. Doesn't include any text of its own.
HtmlInputText and HtmlInputPassword	<input type="text"> and <input type="password">	A single-line text box enabling the user to enter information. Can also be displayed as a password field (which displays bullets instead of characters to hide the user input).
HtmlTextArea	<textarea>	A large text box for typing multiple lines of text.
HtmlInputImage	<input type="image">	Similar to the tag, but inserts a "clickable" image that submits the page. Using server-side code, you can determine exactly where the user clicked in the image—a technique you'll consider later in this chapter.
HtmlInputFile	<input type="file">	A Browse button and text box that can be used to upload a file to your web server, as described in Chapter 17.

(continued)

Table 5-3. (continued)

Class Name	HTML Element	Description
HtmlInputHidden	<input type="hidden">	Contains text information that will be sent to the server when the page is posted back but won't be visible in the browser.
HtmlSelect	<select>	A drop-down or regular list box enabling the user to select an item.
HtmlHead and HtmlTitle	<head> and <title>	Represents the header information for the page, which includes information about the page that isn't actually displayed in the page, such as search keywords and the web page title. These are the only HTML server controls that aren't placed in the <form> section.
HtmlGenericControl	Any other HTML element.	This control can represent a variety of HTML elements that don't have dedicated control classes. For example, if you add the runat="server" attribute to a <div> element, it's provided to your code as an HtmlGenericControl object. You can identify the type of element by reading the TagName property, which stores a string (for example, "div").

Remember, there are two ways to add any HTML server control. You can add it by hand to the markup in the .aspx file (simply insert the ordinary HTML element, and add the runat = "server" attribute). Alternatively, you can drag the control from the HTML section of the Toolbox in Visual Studio, and drop it onto the design surface of a web page. However, this approach doesn't work for every HTML server control, because they don't all appear in the Toolbox.

So far, the currency converter defines three controls, which are instances of the HtmlForm, HtmlInputText, and HtmlInputButton classes, respectively. It's useful to know the class names if you want to look up information about these classes in the class library reference on Microsoft's MSDN website (<http://tinyurl.com/cq63b6m>). Table 5-4 gives a quick overview of some of the most important control properties.

Table 5-4. Important HTML Control Properties

Control	Most Important Properties
HtmlAnchor	HRef, Target
HtmlImage	Src, Alt, Width, Height
HtmlInputCheckBox and HtmlInputRadioButton	Checked
HtmlInputText	Value
HtmlTextArea	Value
HtmlInputImage	Src, Alt
HtmlSelect	Items (collection)
HtmlGenericControl	InnerText and InnerHtml

Adding the Currency Converter Code

To add some functionality to the currency converter, you need to add some ASP.NET code. Web forms are event-driven, which means every piece of code acts in response to a specific event. In the simple currency converter page example, the most useful event occurs when the user clicks the submit button (named Convert). The `HtmlInputButton` allows you to react to this action by handling the `ServerClick` event.

Before you continue, it makes sense to add another control that can display the result of the calculation. In this case, you can use an HTML `<p>`, or paragraph. The `<p>` element is one way to insert a block of formatted text into a web page. Here's the HTML that creates a `<p>` element named Result that starts out with no text:

```
<p style = "font-weight: bold" ID = "Result" runat = "server">/p>
```

The style attribute applies the CSS properties used to format the text. In this example, it merely applies a bold font using the CSS `font-weight` property. If you're not familiar with CSS, don't worry—you'll get a closer review of the essentials in Chapter 12.

The example now has the following four server controls:

- A form (which is represented by the `HtmlForm` object). This is the only control you do not need to access in your code-behind class.
- An input text box named US (`HtmlInputText` object).
- A submit button named Convert (`HtmlInputButton` object).
- A `<p>` element named Result (`HtmlGenericControl` object).

Listing 5-1 shows the revised web page (`CurrencyConverter.aspx`), but leaves out the doctype to save space. Listing 5-2 shows the code-behind class (`CurrencyConverter.aspx.vb` / `CurrencyConverter.aspx.cs`). The code-behind class includes an event handler that reacts when the Convert button is clicked. It calculates the currency conversion and displays the result.

Listing 5-1. CurrencyConverter.aspx

```
<%@ Page Language = "C#" AutoEventWireup = "true"
  CodeFile = "CurrencyConverter.aspx.cs" Inherits = "CurrencyConverter" %>
<!DOCTYPE html>
<html>
<head>
<title> Currency Converter</title>
</head>
<body>
<form runat = "server">
<div>
  Convert: &nbsp;
  <input type = "text" ID = "US" runat = "server" />
  &nbsp; U.S. dollars to Euros.
  <br /> <br />
  <input type = "submit" value = "OK" ID = "Convert" runat = "server"
    OnServerClick = "Convert_ServerClick" />
  <br /> <br />
  <p style = "font-weight: bold" ID = "Result" runat = "server" > </p>
</div>
</form>
</body>
</html>
```

Listing 5-2. CurrencyConverter.aspx.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class CurrencyConverter : System.Web.UI.Page
{
    protected void Convert_ServerClick(object sender, EventArgs e)
    {
        decimal USAmount = Decimal.Parse(US.Value);
        decimal euroAmount = USAmount * 0.85 M;
        Result.InnerText = USAmount.ToString() + " U.S. dollars = ";
        Result.InnerText += euroAmount.ToString() + " Euros.";
    }
}

```

The code-behind class is a typical example of an ASP.NET page. You'll notice the following conventions:

- It starts with *several* *using* statements. This provides access to all the important namespaces. This is a typical first step in any code-behind file.
- The page class is defined with *partial* keyword. That's because your class code is merged with another code file that you never see. This extra code, which ASP.NET generates automatically, defines all the server controls that are used on the page. This allows you to access them by name in your code.
- The page defines a single event handler. This event handler retrieves the value from the text box, converts it to a numeric value, multiplies it by a preset conversion ratio (which would typically be stored in another file or a database), and sets the text of the <p> element. You'll notice that the event handler accepts two parameters (sender and e). This is the .NET standard for all control events. It allows your code to identify the control that sent the event (through the sender parameter) and retrieve any other information that may be associated with the event (through the e parameter). You'll see examples of these advanced techniques in the next chapter, but for now, it's important to realize that you won't be allowed to handle an event unless your event handler has the correct, matching signature.
- The event handler is connected to the control event by using the OnServerClick attribute in the <input> tag for the button. You'll learn more about how this hookup works in the next section.

Note unlike with web controls, you can't create event handlers for HTml server controls by using the Properties window. Instead, you must type the method in by hand, making sure to include the Handles clause at the end of the method definition. One option is to use the two drop-down lists at the top of the code window. To take this approach, choose the method name in the left list (for example, Convert), and then choose the event you want to handle in the list on the right (for example, serverClick). Visual studio will create the corresponding event handler. Instead, you must type the method name in the code window and then add the OnServerClick attribute to the control tag. You must also modify the control tag to connect your event handler. For example, to connect the Convert_ServerClick method named Convert_serverClick, you must add the OnServerClick="Convert_ServerClick" attribute to the control tag.

- The `+=` operator is used to quickly add information to the end of the label, without replacing the existing text.
- The event handler uses `ToString()` to convert the decimal value to text so it can be added to the `InnerText` property. In this particular statement, you don't need `ToString()`, because C# is intelligent enough to realize that you're joining together pieces of text. However, this isn't always the case, so it's best to be explicit about data type conversions.

You can launch this page to test your code. When you enter a value and click the OK button, the page is resubmitted, the event-handling code runs, and the page is returned to you with the conversion details (see Figure 5-3).



Figure 5-3. The ASP.NET currency converter

Event Handling

When the user clicks the Convert button and the page is sent back to the web server, ASP.NET needs to know exactly what code you want to run. To create this relationship and connect an event to an event-handling method, you need to add an attribute in the control tag.

For example, if you want to handle the `ServerClick` method of the `Convert` button, you simply need to set the `OnServerClick` attribute in the control tag with the name of the event handler you want to use:

```
<input type = "submit" value = "OK" ID = "Convert"
OnServerClick = "Convert_ServerClick" runat = "server">
```

ASP.NET controls always use this syntax. When attaching an event handler in the control tag, you use the name of the event preceded by the word `On`. For example, if you want to handle an event named `ServerChange`, you'd set an attribute in the control tag named `OnServerChange`. When you double-click a server control on the design surface, Visual Studio adds the event handler and sets the event attribute to match. (Now that you understand this process, you'll understand the source of a common error. If you double-click a control to create an event handler, but you then delete the event-handling method, you'll end up with an event attribute that points to a nonexistent event. As a result, you'll receive an error the next time you run the page. To fix the problem, you need to remove the event attribute from the control tag.)

Note There's one AsP.nET object that doesn't use this attribute system, because it doesn't have a control class. Instead, AsP.nET connects the web page events based on method names. In other words, if you have a method named Page_load() in your page class, and it has the right signature (it accepts two parameters representing the sender and an EventArgs object), AsP.nET connects this event handler to the Page.load event automatically. This feature is called automatic event wireup.

ASP.NET allows you to use another technique to connect events—you can do it by using code, just as you saw in Chapter 3. For example, here's the code that's required to hook up the ServerClick event of the Convert button using manual event wireup:

```
Convert.ServerClick += this.Convert_ServerClick;
```

If you're using code like this, you'd probably add it to the Page_Load() method so it connects your event handlers when the page is first initialized.

Seeing as Visual Studio handles event wireup, why should ASP.NET developers care that they have two ways to hook up an event handler? Well, most of the time you won't worry about it. But the manual event wireup technique is useful in certain circumstances. The most common example occurs if you want to create a control object and add it to a page dynamically at runtime. In this situation, you can't hook up the event handler through the control tag, because there isn't a control tag. Instead, you need to create the control and attach its event handlers by using code. (The next chapter has an example of how you can use dynamic control creation to fill in a table.)

Behind the Scenes with the Currency Converter

So, what really happens when ASP.NET receives a request for the CurrencyConverter.aspx page? The process unfolds over several steps:

1. The request for the page is sent to the web server. If you're running a live site, the web server is almost certainly IIS, which you'll learn more about in Chapter 26. If you're running the page in Visual Studio, the request is sent to the built-in test server.
2. The web server determines that the .aspx file extension is registered with ASP.NET. If the file extension belonged to another service (as it would for .html files, for example), ASP.NET would never get involved.
3. If this is the first time a page in this application has been requested, ASP.NET automatically creates the application domain. It also compiles all the web page code for optimum performance, and caches the compiled files. If this task has already been performed, ASP.NET will reuse the compiled version of the page.
4. The compiled CurrencyConverter.aspx page acts like a miniature program. It starts firing events (most notably, the Page.Load event). However, you haven't created an event handler for that event, so no code runs. At this stage, everything is working together as a set of in-memory .NET objects.
5. When the code is finished, ASP.NET asks every control in the web page to render itself into the corresponding HTML markup.

Tip In fact, AsP.nET performs a little sleight of hand and may customize the output with additional client-side code such as Javascript or dHTML if it detects that the client browser supports it. In the case of CurrencyConverter.aspx, the output of the page is too simple to require this type of automatic tweaking.

6. The final page is sent to the user, and the application ends.

The description is lengthy, but it's important to start with a good understanding of the fundamentals. When you click a button on the page, the entire process repeats itself. However, in step 4 the ServerClick event fires for the `HtmlInputButton` right after the `Page.Load` event, and your code runs.

Figure 5-4 illustrates the stages in a web page request.

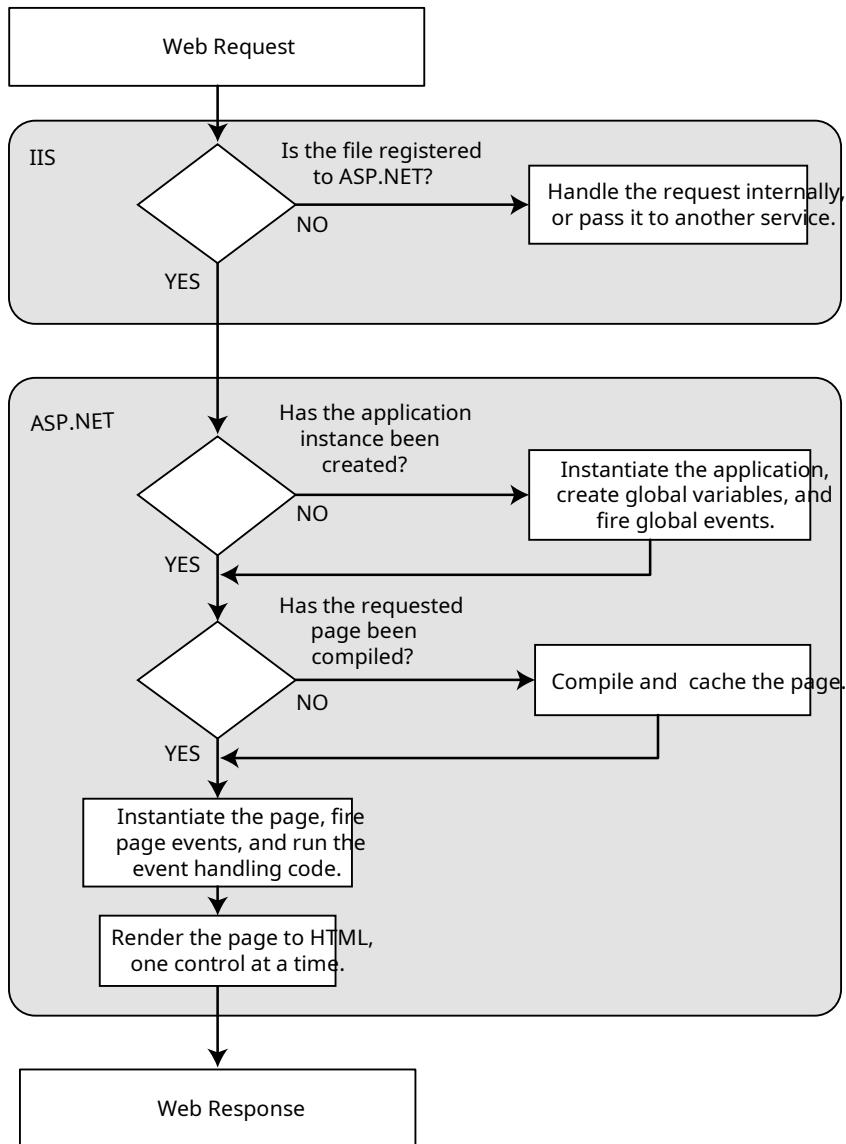


Figure 5-4. The stages in an ASP.NET request

The most important detail is that your code works with objects. The final step is to transform these objects into the appropriate HTML output. A similar conversion from objects to output happens with a Windows program in .NET, but it's so automatic that programmers rarely give it much thought. Also, in those environments, the code always runs locally. In an ASP.NET application, the code runs in a protected environment on the server. The client sees the results only after the web page processing has ended and the web page object has been released from memory.

Error Handling

The currency converter expects that the user will enter a number before clicking the Convert button. If the user enters something else—for example, a sequence of text or special characters that can't be interpreted as a number—an error will occur when the code attempts to use the `Decimal.Parse()` method. In the current version of the currency converter, this error will completely derail the event-handling code. Because this example doesn't include any code to handle errors, ASP.NET will simply send an error page back to the user describing the problem.

In Chapter 7, you'll learn how to deal with errors by catching them, neutralizing them, and informing the user more gracefully. However, even without these abilities, you can rework the code that responds to the `ServerClick` event to avoid potential errors. One good approach is to use the `Decimal.TryParse()` method instead of `Decimal.Parse()`. Unlike `Parse()`, `TryParse()` does not generate an error if the conversion fails—it simply informs you of the problem.

`TryParse()` accepts two parameters. The first parameter is the value you want to convert (in this example, `US.Value`). The second parameter is an output parameter that will receive the converted value (in this case, the variable named `USAmt`). What's special about `TryParse()` is its Boolean return value, which indicates whether the conversion was successful (`true`) or not (`false`).

Here's a revised version of the `ServerClick` event handler that uses `TryParse()` to check for conversion problems and inform the user:

```
protected void Convert_ServerClick(object sender, EventArgs e)
{
    decimal USAmt;

    // Attempt the conversion.
    bool success = Decimal.TryParse(US.Value, out USAmt);

    // Check if it succeeded.
    if (success)
    {
        // The conversion succeeded.
        decimal euroAmount = USAmt * 0.85M;
        Result.InnerText = USAmt.ToString() + " U.S. dollars = ";
        Result.InnerText += euroAmount.ToString() + " Euros.";
    }
    else
    {
        // The conversion failed.
        Result.InnerText = "The number you typed in was not in the " +
            "correct format. Use only numbers.";
    }
}
```

Dealing with error conditions like these is an essential technique in a real-world web page.

Improving the Currency Converter

Now that you've looked at the basic server controls, it might seem that their benefits are fairly minor compared with the cost of learning a whole new system of web programming. In this section, you'll start to extend the currency converter application. You'll see how you can "snap in" additional functionality to the existing program in an elegant, modular way. As the program grows, ASP.NET handles its complexity easily, because it cleanly separates the HTML markup from the code that manipulates it.

Adding Multiple Currencies

The first task is to allow the user to choose a destination currency. In this case, you need to use a drop-down list box. In HTML, a drop-down list is represented by a `<select>` element that contains one or more `<option>` elements. Each `<option>` element corresponds to a separate item in the list.

To reduce the amount of HTML in the currency converter, you can define a drop-down list without any list items by adding an empty `<select>` tag. As long as you ensure that this `<select>` tag is a server control (by giving it an ID and adding the `runat = "server"` attribute), you'll be able to interact with it in code and add the required items when the page loads.

Here's the revised HTML for the `CurrencyConverter.aspx` page:

```
<%@ Page Language = "C#" AutoEventWireup = "true"
  CodeFile = "CurrencyConverter.aspx.cs" Inherits = "CurrencyConverter" %>
<!DOCTYPE html>
<html xmlns =
http://www.w3.org/1999/xhtml
">
<head>
  <title > Currency Converter</title>
</head>
<body>
  <form runat = "server">
    <div>
      Convert: &nbsp;
      <input type = "text" ID = "US" runat = "server" />
      &nbsp; U.S. dollars to &nbsp;
      <select ID = "Currency" runat = "server" />
      <br /> <br />
      <input type = "submit" value = "OK" ID = "Convert"
        OnServerClick = "Convert_ServerClick" runat = "server" />
      <br /> <br />
      <p style = "font-weight: bold" ID = "Result" runat = "server" > </p>
    </div>
  </form>
</body>
</html>
```

The currency list can now be filled using code at runtime. In this case, the ideal event is the `Page.Load` event, which is fired at the beginning of the page-processing sequence. Here's the code you need to add to the `CurrencyConverter` page class:

```
protected void Page_Load(Object sender, EventArgs e)
{
  if (this.IsPostBack == false)
```

```

{
    Currency.Items.Add("Euro");
    Currency.Items.Add("Japanese Yen");
    Currency.Items.Add("Canadian Dollar");
}
}

```

Dissecting the Code . . .

This example illustrates two important points:

- You can use the Items property to get items in a list control. This allows you to append, insert, and remove <option> elements (which represent the items in the list). Remember, when generating dynamic content with a server control, you set the properties, and the control creates the appropriate HTML tags.
- Before adding any items to this list, you need to make sure this is the first time the page is being served to this particular user. Otherwise, the page will continuously add more items to the list or inadvertently overwrite the user's selection every time the user interacts with the page. To perform this test, you check the IsPostBack property of the current Page. In other words, IsPostBack is a property of the CurrencyConverter class, which CurrencyConverter inherits from the generic Page class. If IsPostBack is false, the page is being created for the first time, and it's safe to initialize it.

Storing Information in the List

Of course, if you're a veteran HTML coder, you know that each item in a list also provides a value attribute that you can use to store a value for each item in the list. Because the currency converter uses a short list of hard-coded currencies, this is an ideal place to store the currency conversion rate.

To set the value attribute, you need to create a ListItem object for every item in the list and add that to the HtmlSelect control. The ListItem class provides a constructor that lets you specify the text and value at the same time that you create it, thereby allowing condensed code like this:

```

protected void Page_Load(Object sender, EventArgs e)
{
    if (this.IsPostBack == false)
    {
        // The HtmlSelect control accepts text or ListItem objects.
        Currency.Items.Add(new ListItem("Euros", "0.85"));
        Currency.Items.Add(new ListItem("Japanese Yen", "110.33"));
        Currency.Items.Add(new ListItem("Canadian Dollars", "1.2"));
    }
}

```

To complete the example, you must rewrite the calculation code to take the selected currency into account, as follows:

```

protected void Convert_ServerClick(object sender, EventArgs e)
{
    decimal oldAmount;
    bool success = Decimal.TryParse(US.Value, out oldAmount);
}

```

```

if (success)
{
    // Retrieve the selected ListItem object by its index number.
    ListItem item = Currency.Items[Currency.SelectedIndex];

    decimal newAmount = oldAmount * Decimal.Parse(item.Value);
    Result.InnerText = oldAmount.ToString() + " U.S. dollars = ";
    Result.InnerText += newAmount.ToString() + " " + item.Text;
}
}

```

Figure 5-5 shows the revamped currency converter.

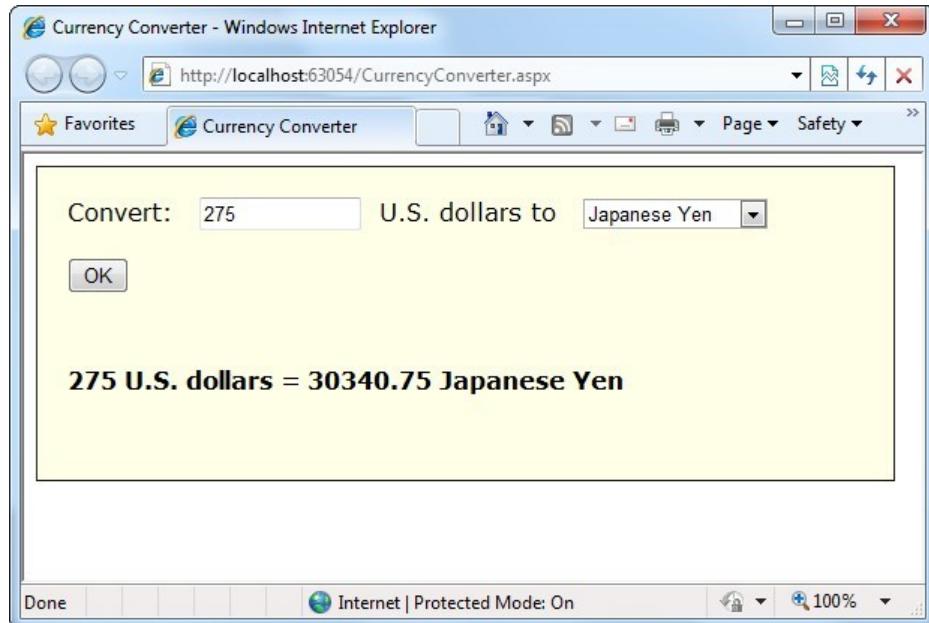


Figure 5-5. The multicurrency converter

All in all, this is a good example of how you can store information in HTML tags by using the value attribute. However, in a more realistic application, you wouldn't actually store the currency rate in the list. Instead, you would just store some sort of unique identifying ID value. Then, when the user submits the page, you would retrieve the corresponding conversion rate from another storage location, such as a database.

Adding Linked Images

Adding other functionality to the currency converter is just as easy as adding a new button. For example, it might be useful for the utility to display a currency conversion rate graph. To provide this feature, the program would need an additional button and image control.

Here's the revised HTML:

```
<%@ Page Language = "C#" AutoEventWireup = "true"
  CodeFile = "CurrencyConverter.aspx.cs" Inherits = "CurrencyConverter" %>
<!DOCTYPE html>
<html>
<head>
  <title> Currency Converter</title>
</head>
<body>
  <form runat = "server">
    <div>
      Convert: &nbsp;
      <input type = "text" ID = "US" runat = "server" />
      &nbsp; U.S. dollars to &nbsp;
      <select ID = "Currency" runat = "server" />
      <br /> <br />
      <input type = "submit" value = "OK" ID = "Convert"
        OnServerClick = "Convert_ServerClick" runat = "server" />
      <b><input type = "submit" value = "Show Graph" ID = "ShowGraph" runat = "server" />
      <br /> <br /></b>
      <img ID = "Graph" src = "" alt = "Currency Graph" runat = "server" />
      <br /> <br />
      <p style = "font-weight: bold" ID = "Result" runat = "server" > </p>
    </div>
  </form>
</body>
</html>
```

As it's currently declared, the image doesn't refer to a picture. For that reason, it makes sense to hide it when the page is first loaded by using this code:

```
protected void Page_Load(Object sender, EventArgs e)
{
  if (this.IsPostBack == false)
  {
    Currency.Items.Add(new ListItem("Euros", "0.85"));
    Currency.Items.Add(new ListItem("Japanese Yen", "110.33"));
    Currency.Items.Add(new ListItem("Canadian Dollars", "1.2"));

    Graph.Visible = false;
  }
}
```

Interestingly, when a server control is hidden, ASP.NET omits it from the final HTML page.

Now you can handle the click event of the new button to display the appropriate picture. The currency converter has three possible picture files—pic0.png, pic1.png, and pic2.png. It chooses one based on the index of the selected item (so it uses pic0.png if the first currency is selected, pic1.png for the second, and so on). Here's the code that shows the chart:

```
protected void ShowGraph_ServerClick(Object sender, EventArgs e)
{
  Graph.Src = "Pic" + Currency.SelectedIndex.ToString() + ".png";
  Graph.Visible = true;
}
```

You need to make sure you link to the event handler through the button, so modify the `<input>` element for the button as follows:

```
<input type = "submit" value = "Show Graph" ID = "ShowGraph"
OnServerClick = "ShowGraph_ServerClick" runat = "server" />
```

You'll also need to change the graph picture when the currency is changed, using this line of code in the `Convert_ServerClick()` method:

```
Graph.Src = "Pic" + Currency.SelectedIndex.ToString() + ".png";
```

If this were a more complex task (for example, one that required multiple lines of code), it would make sense to put it in a separate private method. You could then call that method from both the `ShowGraph_ServerClick()` and `Convert_ServerClick()` event-handling methods, making sure you don't duplicate your code.

Already the currency converter is beginning to look more interesting, as shown in Figure 5-6.

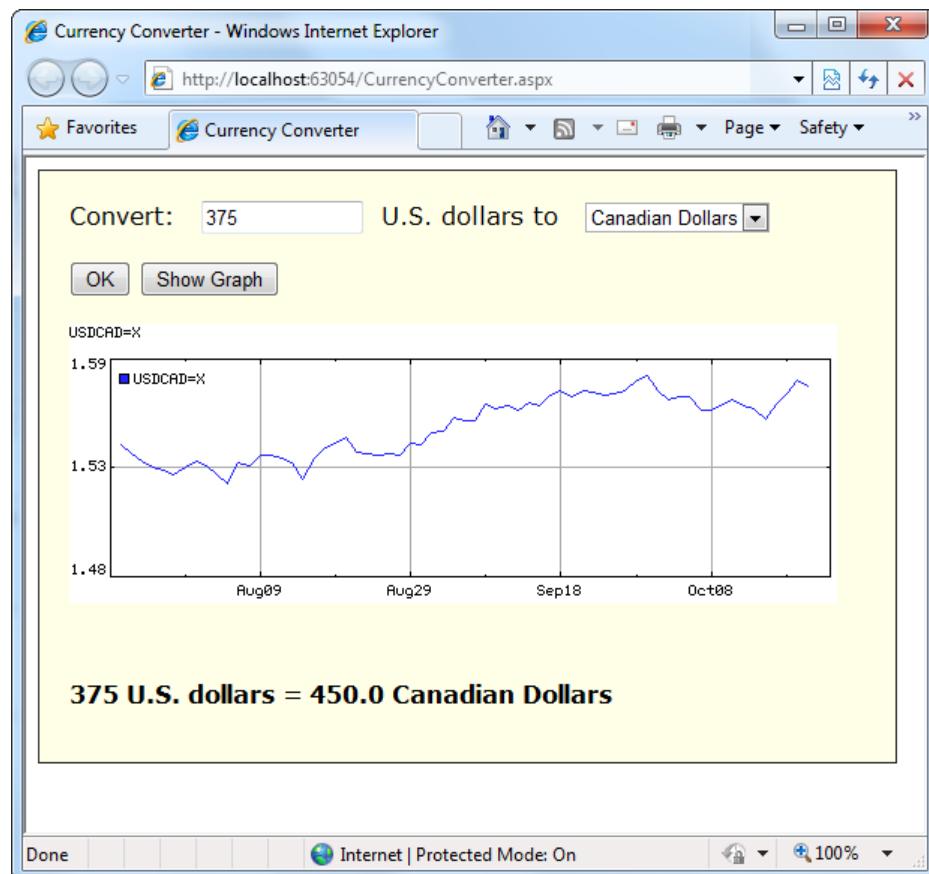


Figure 5-6. The currency converter with an image control

Tip Of course, in a real currency converter, a fixed picture wouldn't do. You could pull a recent image database, but most likely you'd want to generate the currency chart yourself, using server-side drawing. Chapter 11 explains how you would go about this somewhat tedious task.

Setting Styles

In addition to a limited set of properties, each HTML control also provides access to the CSS attributes through its Style collection. To use this collection, you need to specify the name of the CSS style attribute and the value you want to assign to it. Here's the basic syntax:

```
ControlName.Style["AttributeName"] = "AttributeValue";
```

For example, you could use this technique to emphasize an invalid entry in the currency converter with the color red. In this case, you would also need to reset the color to its original value for valid input, because the control uses view state to remember all its settings, including its style properties:

```
protected void Convert_ServerClick(object sender, EventArgs e)
{
    decimal oldAmount;
    bool success = Decimal.TryParse(US.Value, out oldAmount);

    if ((oldAmount <= 0) || (success == false))
    {
        Result.Style["color"] = "Red";
        Result.InnerText = "Specify a positive number";
    }
    else
    {
        Result.Style["color"] = "Black";

        // Retrieve the selected ListItem object by its index number.
        ListItem item = Currency.Items[Currency.SelectedIndex];

        decimal newAmount = oldAmount * Decimal.Parse(item.Value);
        Result.InnerText = oldAmount.ToString() + " U.S. dollars = ";
        Result.InnerText += newAmount.ToString() + " " + item.Text;
    }
}
```

Tip The style collection sets the style attribute in the HTml tag with a list of formatting options such as family, size, and color. You'll learn more in Chapter 12. but if you aren't familiar with Css styles, you don't use them. Instead, you could use web controls, which provide higher-level properties that allow you to control their appearance and automatically create the appropriate style attributes. You'll learn about web controls in chapter.

This concludes the simple currency converter application, which now boasts automatic calculation, linked images, and dynamic formatting. In the following sections, you'll look at the building blocks of ASP.NET interfaces more closely.

Taking a Deeper Look at HTML Control Classes

Related classes in the .NET Framework use inheritance to share functionality. For example, every HTML control inherits from the base class `HtmlControl`. The `HtmlControl` class provides essential features that every HTML server control uses. Figure 5-7 shows the inheritance diagram.

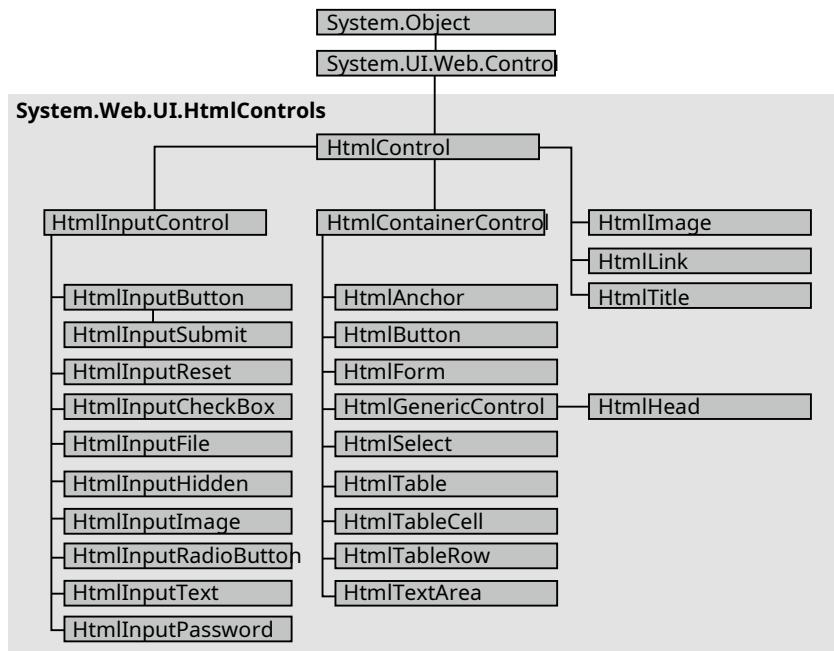


Figure 5-7. HTML control inheritance

This section dissects the ASP.NET classes that are used for HTML server controls. You can use this material to help understand the common elements that are shared by all HTML controls. For the specific details about each HTML control, you can refer to the class library reference on the MSDN website at <http://tinyurl.com/cq63b6m>.

HTML server controls generally provide properties that closely match their tag attributes. For example, the `HtmlImage` class provides `Align`, `Alt`, `Border`, `Src`, `Height`, and `Width` properties. For this reason, users who are familiar with HTML syntax will find that HTML server controls are the most natural fit. Users who aren't as used to HTML will probably find that web controls (described in the next chapter) have a more intuitive set of properties.

HTML Control Events

HTML server controls also provide one of two possible events: `ServerClick` or `ServerChange`.

The `ServerClick` event is simply a click that's processed on the server side. It's provided by most button controls, and it allows your code to take immediate action. For example, consider the `HtmlAnchor` control, which

is the server control that represents the common HTML hyperlink (the `<a>` element). There are two ways to use the `HtmlAnchor` control. One option is to set its `HtmlAnchor.HRef` property to a URL, in which case the hyperlink will behave exactly like the ordinary HTML `<a>` element (the only difference being that you can set the URL dynamically in your code). The other option is to handle the `HtmlAnchor.ServerClick` event. In this case, when the link is clicked, it will actually post back the page, allowing your code to run. The user won't be redirected to a new page unless you provide extra code to forward the request.

The `ServerChange` event responds when a change has been made to a text or selection control. This event isn't as useful as it appears, because it doesn't occur until the page is posted back (for example, after the user clicks a submit button). At this point, the `ServerChange` event occurs for all changed controls, followed by the appropriate `ServerClick`. The `Page.Load` event is the first to fire, but you have no way to know the order of events for other controls.

Table 5-5 shows which controls provide a `ServerClick` event and which ones provide a `ServerChange` event.

Table 5-5. HTML Control Events

Event	Controls That Provide It
<code>ServerClick</code>	<code>HtmlAnchor</code> , <code>HtmlButton</code> , <code>HtmlInputButton</code> , <code>HtmlInputImage</code> , <code>HtmlInputReset</code>
<code>ServerChange</code>	<code>HtmlInputText</code> , <code>HtmlInputCheckBox</code> , <code>HtmlInputRadioButton</code> , <code>HtmlInputHidden</code> , <code>HtmlSelect</code> , <code>HtmlTextArea</code>

Advanced Events with the `HtmlInputImage` Control

Chapter 4 introduced the .NET event standard, which dictates that every event should pass exactly two pieces of information. The first parameter identifies the object (in this case, the control) that fired the event. The second parameter is a special object that can include additional information about the event.

In the examples you've looked at so far, the second parameter (`e`) has always been used to pass an empty `System.EventArgs` object. This object doesn't contain any additional information—it's just a glorified placeholder. Here's one such example:

```
protected void Convert_ServerClick(Object sender, EventArgs e)
{. . .}
```

In fact, only one HTML server control sends additional information: the `HtmlInputImage` control. It sends an `ImageEventArgs` object (from the `System.Web.UI` namespace) that provides `X` and `Y` properties representing the location where the image was clicked. You'll notice that the definition for the `HtmlInputImage`.`ServerClick` event handler is a little different from the event handlers used with other controls:

```
protected void ImgButton_ServerClick(Object sender, ImageEventArgs e)
{. . .}
```

Using this additional information, you can replace multiple button controls and image maps with a single, intelligent `HtmlInputImage` control.

Here's the markup you need to create the `HtmlInputImage` control for this example:

```
<input type = "image" ID = "ImgButton" runat = "server" src = "button.png"
OnServerClick = "ImgButton_ServerClick" />
```

The sample `ImageTest.aspx` page shown in Figure 5-8 puts this feature to work with a simple graphical button. Depending on whether the user clicks the button border or the button surface, a different message is displayed.

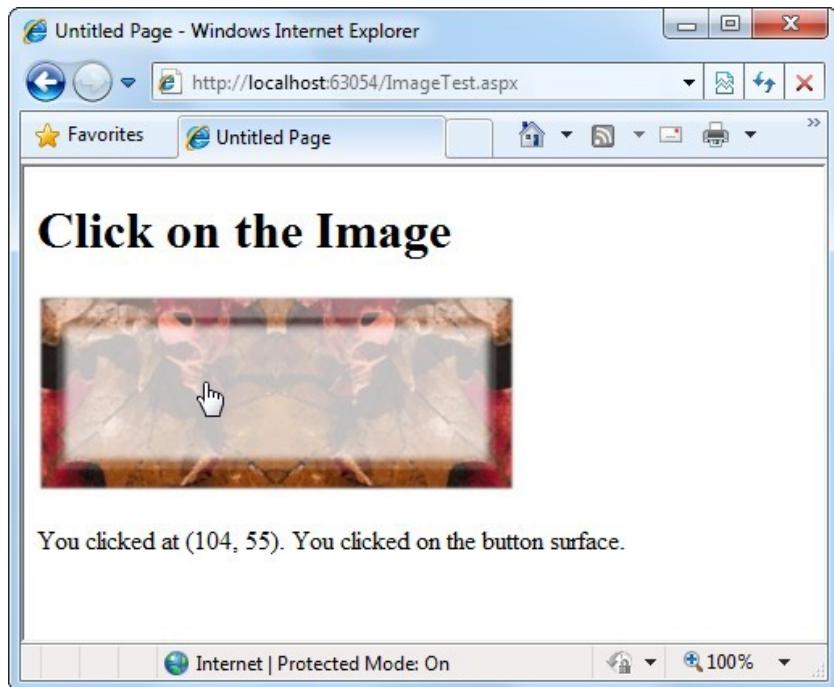


Figure 5-8. Using an `HtmlInputImage` control

The page code examines the click coordinates provided by the `ImageClickEventArgs` object and displays them in another control. Here's the page code you need:

```
public partial class ImageTest : System.Web.UI.Page
{
    protected void ImgButton_ServerClick(Object sender,
        ImageClickEventArgs e)
    {
        Result.InnerText = "You clicked at (" + e.X.ToString() +
            ", " + e.Y.ToString() + "). ";

        if ((e.Y < 100) && (e.Y > 20) && (e.X > 20) && (e.X < 275))
        {
            Result.InnerText += "You clicked on the button surface.";
        }
        else
        {
            Result.InnerText += "You clicked the button border.";
        }
    }
}
```

The HtmlControl Base Class

Every HTML control inherits from the base class `HtmlControl`. This relationship means that every HTML control will support a basic set of properties and features. Table 5-6 shows these properties.

Table 5-6. *HtmlControl Properties*

Property	Description
Attributes	Provides a collection of all the attributes that are set in the control tag, and their values. Rather than reading or setting an attribute through the <code>Attributes</code> , it's better to use the corresponding property in the control class. However, the <code>Attributes</code> collection is useful if you need to add or configure a custom attribute or an attribute that doesn't have a corresponding property.
Controls	Provides a collection of all the controls contained inside the current control. (For example, a <code><div></code> server control could contain an <code><input></code> server control.) Each object is provided as a generic <code>System.Web.UI.Control</code> object so that you may need to cast the reference to access control-specific properties.
Disabled	Disables the control when set to true, thereby ensuring that the user cannot interact with it, and its events will not be fired.
EnableViewState	Disables the automatic state management for this control when set to false. In this case, the control will be reset to the properties and formatting specified in the control tag every time the page is posted back. If this is set to true (the default), the control stores its state in a hidden input field on the page, thereby ensuring that any changes you make in code are remembered. (For more information, see the "View State" section earlier in this chapter.)
Page	Provides a reference to the web page that contains this control as a <code>System.Web.UI.Page</code> object.
Parent	Provides a reference to the control that contains this control. If the control is placed directly on the page (rather than inside another control), it will return a reference to the page object.
Style	Provides a collection of CSS style properties that can be used to format the control.
TagName	Indicates the name of the underlying HTML element (for example, <code>img</code> or <code>div</code>).
Visible	Hides the control when set to false and will not be rendered to the final HTML page that is sent to the client.

To set the initial value of a property, you can configure the control in the `Page.Load` event handler, or you can adjust the control tag in the `.aspx` file by adding attributes. Note that the `Page.Load` event occurs after the page is initialized with the default values and the tag settings. This means your code can override the properties set in the tag (but not vice versa).

The `HtmlContainerControl` Class

Any HTML control that requires a closing tag inherits from the `HtmlContainer` class (which in turn inherits from the more basic `HtmlControl` class). For example, elements such as `<a>`, `<form>`, and `<div>` always use a closing tag, because they can contain other HTML elements. On the other hand, elements such as `` and `<input>` are used only as stand-alone tags. Thus, the `HtmlAnchor`, `HtmlForm`, and `HtmlGenericControl` classes inherit from `HtmlContainerControl`, while `HtmlInputImage` and `HtmlInputButton` do not.

The `HtmlContainer` control adds two properties to those defined in `HtmlControl`, as described in Table 5-7.

Table 5-7. *HtmlContainerControl Properties*

Property	Description
InnerHtml	The HTML content between the opening and closing tags of the control. Special characters that are set through this property will not be converted to the equivalent HTML entities. This means you can use this property to apply formatting with nested tags such as <code></code> , <code><i></code> , and <code><h1></code> .
InnerText	The text content between the opening and closing tags of the control. Special characters will be automatically converted to HTML entities and displayed as text (for example, the less-than character (<code><</code>) will be converted to <code>&lt;</code> and will be displayed as <code><</code> in the web page). This means you can't use HTML tags to apply additional formatting with this property. The simple currency converter page uses the <code>InnerText</code> property to enter results into a <code><p></code> element.

The `HtmlInputControl` Class

The `HtmlInputControl` class inherits from `HtmlControl` and adds some properties (shown in Table 5-8) that are used for the `<input>` element. As you've already learned, the `<input>` element can represent different controls, depending on the `type` attribute. The `<input type = "text">` element is a text box, and `<input type = "submit">` is a button.

Table 5-8. *HtmlInputControl Properties*

Property	Description
Type	Provides the type of input control. For example, a control based on <code><input type = "file"></code> would return <code>file</code> for the <code>type</code> property.
Value	Returns the contents of the control as a string. In the simple currency converter, this property allowed the code to retrieve the information entered in the text input control.

Using the Page Class

One control you haven't considered in detail yet is the `Page` class. As explained in the previous chapter, every web page is a custom class that inherits from `System.Web.UI.Page`. By inheriting from this class, your web page class acquires a number of properties and methods that your code can use. These include properties for enabling caching, validation, and tracing, which are discussed throughout this book. Table 5-9 provides an overview of some of the more fundamental `Page` class properties, which you'll use throughout this book.

Table 5-9. Basic Page Properties

Property	Description
IsPostBack	This Boolean property indicates whether this is the first time the page is being run (false) or whether the page is being resubmitted in response to a control event, typically with stored view state information (true). You'll usually check this property in the Page.Load event handler to ensure that your initial web page initialization is performed only once.
EnableViewState	When set to false, this overrides the EnableViewState property of the contained controls, thereby ensuring that no controls will maintain state information.
Application	This collection holds information that's shared between all users in your website. For example, you can use the Application collection to count the number of times a page has been visited. You'll learn more in Chapter 8.
Session	This collection holds information for a single user, so it can be used in different pages. For example, you can use the Session collection to store the items in the current user's shopping basket on an e-commerce website. You'll learn more in Chapter 8.
Cache	This collection allows you to store objects that are time-consuming to create so they can be reused in other pages or for other clients. This technique, when implemented properly, can improve performance of your web pages. Chapter 23 discusses caching in detail.
Request	This refers to an HttpRequest object that contains information about the current web request. You can use the HttpRequest object to get information about the user's browser, although you'll probably prefer to leave these details to ASP.NET. You'll use the HttpRequest object to transmit information from one page to another with the query string in Chapter 8.
Response	This refers to an HttpResponse object that represents the response ASP.NET will send to the user's browser. You'll use the HttpResponse object to create cookies in Chapter 8, and you'll see how it allows you to redirect the user to a different web page later in this chapter.
Server	This refers to an HttpServerUtility object that allows you to perform a few miscellaneous tasks. For example, it allows you to encode text so that it's safe to place it in a URL or in the HTML markup of your page. You'll learn more about these features in this chapter.
User	If the user has been authenticated, this property will be initialized with user information. Chapter 19 describes this property in more detail.

Many of the Page properties provide complete objects. For example, the Page.Response property allows you to access the Response object anywhere in your page, and the Page.Server property allows you to access the Server object. These two objects are fundamentally important for two basic tasks that are explained in the following sections. First, you can use the Response and Server objects to send the user from one page to another, which is a key part of any ASP.NET application. Second, you can use the Server object to encode text that may contain special characters so it can be inserted into web page HTML.

Sending the User to a New Page

In the currency converter example, everything took place in a single page. In a more typical website, the user will need to surf from one page to another to perform different tasks or complete a single operation.

There are several ways to transfer a user from one page to another. One of the simplest is to use an ordinary `<a>` anchor element, which turns a portion of text into a hyperlink. In this example, the word *here* is a link to another page:

```
Click <a href = "newpage.aspx" > here</a > to go to newpage.aspx.
```

Another option is to send the user to a new page by using code. This approach is useful if you want to use your code to perform some other work before you redirect the user. It's also handy if you need to use code to decide where to send the user. For example, if you create a sequence of pages for placing an order, you might send existing customers straight to the checkout, while new visitors are redirected to a registration page.

To perform redirection in code, you first need a control that causes the page to be posted back. In other words, you need an event handler that reacts to the `ServerClick` event of a control such as `HtmlInputButton` or `HtmlAnchor`. When the page is posted back and your event handler runs, you can use the `HttpResponse.Redirect()` method to send the user to the new page.

Remember, you can get access to the current `HttpResponse` object through the `Page.Response` property. Here's an example that sends the user to a different page in the same website directory:

```
Response.Redirect("newpage.aspx");
```

When you use the `Redirect()` method, ASP.NET immediately stops processing the page and sends a redirect message back to the browser. Any code that occurs after the `Redirect()` call won't be executed. When the browser receives the redirect message, it sends a request for the new page.

You can use the `Redirect()` method to send the user to any type of page. You can even send the user to another website by using an absolute URL (a URL that starts with `http://`), as shown here:

```
Response.Redirect("http://www.prosetech.com");
```

ASP.NET gives you one other option for sending the user to a new page. You can use the `HttpServerUtility.Transfer()` method instead of `Response.Redirect()`. An `HttpServerUtility` object is provided through the `Page.Server` property, so your redirection code would look like this:

```
Server.Transfer("newpage.aspx");
```

The advantage of using the `Transfer()` method is that it doesn't involve the browser. Instead of sending a redirect message back to the browser, ASP.NET simply starts processing the new page as though the user had originally requested that page. This behavior saves a bit of time, but it also introduces some significant limitations. You can't use `Transfer()` to send the user to another website or to a non-ASP.NET page (such as an HTML page). The `Transfer()` method allows you to jump only from one ASP.NET page to another, in the same web application. Furthermore, when you use `Transfer()`, the user won't have any idea that another page has taken over, because the browser will still show the original URL. This can cause a problem if you want to support browser bookmarks. On the whole, it's much more common to use `HttpResponse.Redirect()` than `HttpServerUtility.Transfer()`.

Working with HTML Encoding

As you already know, certain characters have a special meaning in HTML. For example, angle brackets (`<>`) are always used to create tags. This can cause problems if you want to use these characters as part of the content of your web page.

For example, imagine you want to display this text on a web page:

Enter a word < here>

If you try to write this information to a page or place it inside a control, you end up with this instead:

Enter a word

The problem is that the browser has tried to interpret the <here> as an HTML tag. A similar problem occurs if you actually use valid HTML tags. For example, consider this text:

To bold text use the tag.

Not only will the text not appear, but the browser will interpret it as an instruction to make the text that follows bold. To circumvent this automatic behavior, you need to convert potential problematic values to their HTML equivalents. For example, < becomes < in your final HTML page, which the browser displays as the <character. Table 5-10 lists some special characters that need to be encoded.

Table 5-10. Common HTML Special Characters

Result	Description	Encoded Entity
	Nonbreaking space	
<	Less-than symbol	<
>	Greater-than symbol	>
&	Ampersand	&
"	Quotation mark	"

You can perform this transformation on your own, or you can circumvent the problem by using the InnerText property of an HTML server control. When you set the contents of a control by using InnerText, any illegal characters are automatically converted into their HTML equivalents. However, this won't help if you want to set a tag that contains a mix of embedded HTML tags and encoded characters. It also won't be of any use for controls that don't provide an InnerText property, such as the Label web control you'll examine in the next chapter. In these cases, you can use the `HttpServerUtility.HtmlEncode()` method to replace the special characters. (Remember, an `HttpServerUtility` object is provided through the `Page.Server` property.)

Here's an example:

```
// Will output as "Enter a word &lt;here&gt;" in the HTML file, but the
// browser will display it as "Enter a word < here >".
ctrl.InnerHtml = Server.HtmlEncode("Enter a word < here >");
```

Or consider this example, which mingles real HTML tags with text that needs to be encoded:

```
ctrl.InnerHtml = "To < b > bold</b > text use the ";
ctrl.InnerHtml += Server.HtmlEncode("< b >") + " tag. ";
```

Figure 5-9 shows the results of successfully and incorrectly encoding special HTML characters. You can refer to the HtmlEncodeTest.aspx page included with the examples for this chapter.

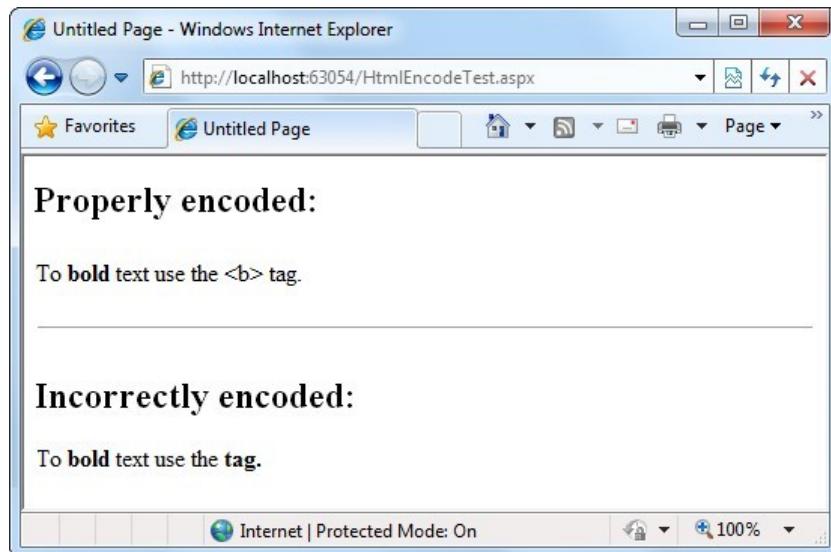


Figure 5-9. Encoding special HTML characters

The `HtmlEncode()` method is particularly useful if you're retrieving values from a database and you aren't sure whether the text is valid HTML. You can use the `HttpUtility.HtmlDecode()` method to revert the text to its normal form if you need to perform additional operations or comparisons with it in your code.

Along with the `HtmlEncode()` and `HttpUtility.HtmlDecode()` methods, the `HttpServerUtility` class also includes `UrlEncode()` and `UrlDecode()` methods. Much as `HtmlEncode()` allows you to convert text to valid HTML with no special characters, `UrlEncode()` allows you to convert text into a form that can be used in a URL. This technique is particularly useful if you want to pass information from one page to another by tacking it onto the end of the URL. You'll see this technique demonstrated in Chapter 8.

Using Application Events

In this chapter, you've seen how ASP.NET controls fire events that you can handle in your code. Although server controls are the most common source of events, there's another type of event that you'll occasionally encounter: *application events*. Application events aren't nearly as important in an ASP.NET application as the events fired by server controls, but you might use them to perform additional processing tasks. For example, by using application events, you can write logging code that runs every time a request is received, no matter what page is being requested. Basic ASP.NET features such as session state and authentication use application events to plug into the ASP.NET processing pipeline.

You can't handle application events in the code-behind for a web form. Instead, you need the help of another ingredient: the `global.asax` file.

The global.asax File

The global.asax file allows you to write code that responds to global application events. These events fire at various points during the lifetime of a web application, including when the application domain is first created (when the first request is received for a page in your website folder).

To add a global.asax file to an application in Visual Studio, choose Website ➢ Add New Item, and select the Global Application Class file type. Then click OK. (If you already have a global.asax file in your project, you won't see the Global Application Class file type, because Visual Studio is smart enough to realize that a single website can't have two.)

The global.asax file looks similar to a normal .aspx file, except that it can't contain any HTML or ASP.NET tags. Instead, it contains event handlers that respond to application events. When you add the global.asax file, Visual Studio inserts several ready-made application event handlers. You simply need to fill in some code.

For example, the following global.asax file reacts to the EndRequest event, which happens just before the page is sent to the user:

```
<%@ Application Language = "C#" %>

<script language = "c#" runat = "server">
    protected void Application_EndRequest(object sender, EventArgs e)
    {
        Response.Write("< hr />This page was served at " +
            DateTime.Now.ToString());
    }
</script>
```

This event handler uses the Write() method of the built-in Response object to write a footer at the bottom of the page with the date and time that the page was created (see Figure 5-10).

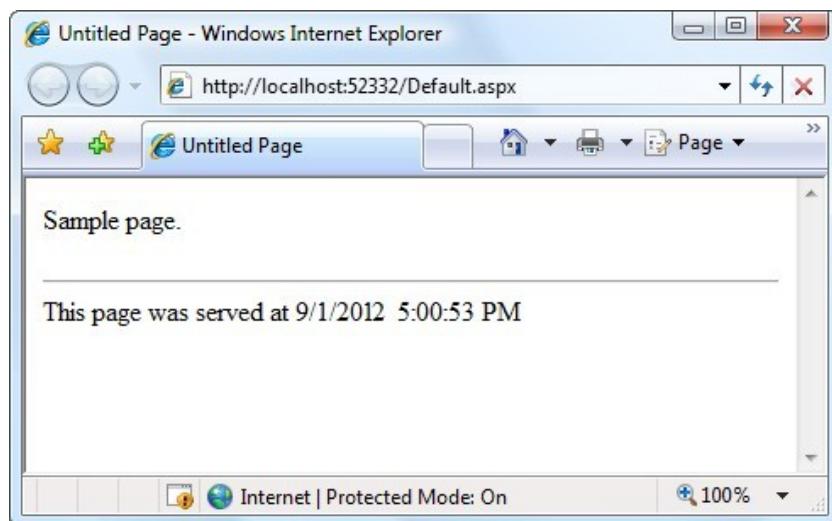


Figure 5-10. A sample page with an automatic footer

Each ASP.NET application can have one global.asax file. After you place it in the appropriate website directory, ASP.NET recognizes it and uses it automatically. For example, if you add the global.asax file shown previously to a web application, every web page in that application will include a footer.

Note This technique—responding to application events and using the `Response.Write()` method—isn’t the best way to add a footer to the pages in your website. A better approach is to add a user control that creates the footer (Chapter 11) or define a master page template that includes a footer (Chapter 12).

Additional Application Events

`Application.EndRequest` is only one of more than a dozen events you can respond to in your code. To create a different event handler, you simply need to create a subroutine with the defined name. Table 5-11 lists some of the most common application events that you’ll use.

Table 5-11. Basic Application Events

Event-Handling Method	Description
<code>Application_Start()</code>	Occurs when the application starts, which is the first time it receives a request from any user. It doesn’t occur on subsequent requests. This event is commonly used to create or cache some initial information that will be reused later.
<code>Application_End()</code>	Occurs when the application is shutting down, generally because the web server is being restarted. You can insert cleanup code here.
<code>Application_BeginRequest()</code>	Occurs with each request the application receives, just before the page code is executed.
<code>Application_EndRequest()</code>	Occurs with each request the application receives, just after the page code is executed.
<code>Session_Start()</code>	Occurs whenever a new user request is received and a session is started. Sessions are discussed in detail in Chapter 8.
<code>Session_End()</code>	Occurs when a session times out or is programmatically ended. This event is raised only if you are using in-process session-state storage (the <code>InProc</code> mode, not the <code>StateServer</code> or <code>SQLServer</code> modes).
<code>Application_Error()</code>	Occurs in response to an unhandled error. You can find more information about error handling in Chapter 7.

Configuring an ASP.NET Application

The last topic you’ll consider in this chapter is the ASP.NET configuration file system.

Every web application includes a `web.config` file that configures fundamental settings—everything from the way error messages are shown to the security settings that lock out unwanted visitors. You’ll consider the settings in the `web.config` file throughout this book. (And there are many more settings that you *won’t* consider in this book, because they’re used much more rarely.)

The ASP.NET configuration files have several key advantages:

They are never locked: You can update `web.config` settings at any point, even while your application is running. If there are any requests currently under way, they’ll continue to use the old settings, while new requests will get the changed settings right away.

They are easily accessed and replicated: Provided you have the appropriate network rights, you can change a web.config file from a remote computer. You can also copy the web.config file and use it to apply identical settings to another application or another web server that runs the same application in a web farm scenario.

The settings are easy to edit and understand: The settings in the web.config file are human-readable, which means they can be edited and understood without needing a special configuration tool.

In the following sections, you'll get a high-level overview of the web.config file and learn how ASP.NET's configuration system works.

Working with the web.config File

The web.config file uses a predefined XML format. The entire content of the file is nested in a root <configuration> element. Inside this element are several more subsections, some of which you'll never change, and others which are more important.

Here's the basic skeletal structure of the web.config file:

```
<?xml version = "1.0" ?>
<configuration>
  <appSettings> . . . </appSettings>
  <connectionStrings> . . . </connectionStrings>
  <system.web> . . . </system.web>
</configuration>
```

Note that the web.config file is case-sensitive, like all XML documents, and starts every setting with a lowercase letter. This means you cannot write <AppSettings> instead of <appSettings> .

Tip To learn more about Xml, the format used for the web.config file, you can refer to Chapter 18.

As a web developer, there are three sections in the web.config file that you'll work with. The <appSettings> section allows you to add your own miscellaneous pieces of information. You'll learn how to use it in the next section. The <connectionStrings> section allows you to define the connection information for accessing a database. You'll learn about this section in Chapter 14. Finally, the <system.web> section holds every ASP.NET setting you'll need to configure.

Inside the <system.web> element are separate elements for each aspect of website configuration. You can include as few or as many of these as you want. For example, if you want to control how ASP.NET's security works, you'd add the <authentication> and <authorization> sections.

When you create a new website, it starts with no web.config file. But when you run that website in Visual Studio and choose to enable debugging, Visual Studio creates a basic web.config file with a small number of settings. It looks like this:

```
<?xml version = "1.0"?>
<configuration>
  <appSettings>
    <add key = "aspnet:UseTaskFriendlySynchronizationContext" value = "true"/>
    <add key = "ValidationSettings:UnobtrusiveValidationMode" value = "WebForms"/>
  </appSettings>
  <system.web>
    <compilation debug = "true" targetFramework = "4.5"/>
    <httpRuntime requestValidationMode = "4.5" targetFramework = "4.5"
      encoderType = "..."/>
```

```
<pages controlRenderingCompatibilityVersion = "4.5"/>
<machineKey compatibilityMode = "Framework45"/>
</system.web>
</configuration>
```

This configuration file includes several details. First, there are two hard-coded values (in the `<appSettings>` section) that your code, or other parts of ASP.NET, can pay attention to. You'll learn how that works in a moment.

Most important is the `<compilation>` element, which uses two key attributes to specify two key details:

debug: This attribute tells ASP.NET whether to compile the application in debug mode so you can use Visual Studio's debugging tools. As you saw in Chapter 4, Visual Studio asks whether you want to switch on debugging mode the first time you run a new application (and you should always choose yes).

Note The disadvantage of debugging support is that it slows down your application ever so slightly, and slowdowns can add up to big bottlenecks in a heavily trafficked web application in the real world. For that reason, you should remove the `debug = "true"` attribute when you're ready to deploy your application to a live web server.

targetFramework: This attribute tells Visual Studio what version of .NET you're planning to use on your web server (which determines the features you'll have access to in your web application). As you learned in Chapter 4, you set the target framework when you create your website, and you can change it at any time after. The rest of the configuration details—the `<httpRuntime>`, `<pages>`, and `<machineKey>` elements—are simply there for backward compatibility. In the autogenerated web.config file, shown earlier in this section, all these details simply tell ASP.NET to use the latest behavior introduced with .NET 4.5. (Although the changes from .NET 4 to .NET 4.5 are small, minor differences could trip up certain applications. These configuration settings are a sort of safety valve that allows developers to get back to the way things were if a new tweak or fix changes the behavior of the application when they move it to ASP.NET 4.5.)

The `<system.web>` section is also home for many more important settings that aren't used in the autogenerated web.config file. You'll consider the different elements that you can add to the `<system.web>` section throughout this book.

Understanding Nested Configuration

ASP.NET uses a multilayered configuration system that allows you to set settings at different levels.

Every web server starts with some basic settings that are defined in a directory such as `c:\Windows\Microsoft.NET\Framework\[Version]\Config`. The `[Version]` part of the directory name depends on the version of .NET that you have installed. At the time of this writing, the latest build was v4.0.30319, which makes the full folder name `c:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config`.

Note don't be confused by the fact that the version number in the folder seems to indicate .NET 4 rather than .NET 4.5. This harmless quirk is a side-effect of the way .NET 4.5 was released. Technically, .NET 4.5 is an *in-place update* that extends .NET 4 without replacing it. Thus, .NET 4.5 keeps using the same configuration folder.

The Config folder contains two files, named `machine.config` and `web.config`. Generally, you won't edit either of these files by hand, because they affect the entire computer. Instead, you'll configure the `web.config` file in your

web application folder. Using that file, you can set additional settings or override the defaults that are configured in the two system files.

More interestingly, you can use different settings for different parts of your application. To use this technique, you need to create additional subdirectories inside your virtual directory. These subdirectories can contain their own web.config files with additional settings.

Subdirectories inherit web.config settings from the parent directory. For example, imagine you create a website in the directory c:\ASP.NET\TestWeb. Inside this directory, you create a folder named Secure. Pages in the c:\ASP.NET\TestWeb\Secure directory can acquire settings from three files, as shown in Figure 5-11.

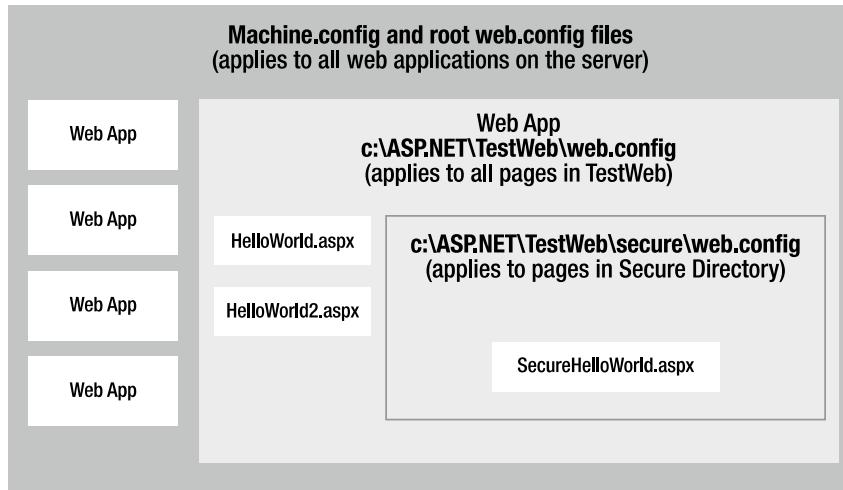


Figure 5-11. Configuration inheritance

Any machine.config or web.config settings that aren't explicitly overridden in the c:\ASP.NET\TestWeb\Secure\web.config file will still apply to the SecureHelloWorld.aspx page. In this way, subdirectories can specify just a small set of settings that differ from the rest of the web application. One reason you might want to use multiple directories in an application is to apply different security settings. Files that need to be secured would then be placed in a dedicated directory with a web.config file that defines more-stringent security settings.

Storing Custom Settings in the web.config File

ASP.NET also allows you to store your own settings in the web.config file, in an element called `<appSettings>`. Note that the `<appSettings>` element is nested in the root `<configuration>` element. Here's the basic structure:

```

<?xml version = "1.0" ?>
<configuration>
  <appSettings>
    <!-- Custom application settings go here. -->
  </appSettings>
  <system.web>
    <!-- ASP.NET Configuration sections go here. -->
  </system.web>
</configuration>
  
```

Note This example adds a comment in the place where you'd normally find additional settings. XML comments are bracketed with the <!-- and --> character sequences. You can also use XML comments to temporarily disable a setting in a configuration file.

The custom settings that you add are written as simple string variables. You might want to use a special web.config setting for several reasons:

To centralize an important setting that needs to be used in many different pages: For example, you could create a variable that stores a database query. Any page that needs to use this query can then retrieve this value and use it.

To make it easy to quickly switch between different modes of operation: For example, you might create a special debugging variable. Your web pages could check for this variable and, if it's set to a specified value, output additional information to help you test the application.

To set some initial values: Depending on the operation, the user might be able to modify these values, but the web.config file could supply the defaults.

You can enter custom settings by using an <add> element that identifies a unique variable name (key) and the variable contents (value). The following example adds a variable that defines a file path where important information is stored:

```
<appSettings>
<add key = "DataFilePath"
    value = "e:\NetworkShare\Documents\WebApp\Shared" />
</appSettings>
```

You can add as many application settings as you want, although this example defines just one.

You can create a simple test page to query this information and display the results, as shown in the following example (which is provided with the sample code as ShowSettings.aspx and ShowSettings.aspx.cs). You retrieve custom application settings from web.config by key name, using the WebConfigurationManager class, which is found in the System.Web.Configuration namespace. This class provides a static property called AppSettings with a collection of application settings.

```
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Configuration;
public partial class ShowSettings : System.Web.UI.Page
{
    protected void Page_Load()
    {
        Results.InnerHtml = "This app will look for data in the directory:<br />";
        Results.InnerHtml += "<b>" + WebConfigurationManager.AppSettings["DataFilePath"];
        Results.InnerHtml += "</b> ";
    }
}
```

Tip notice that this code formats the text by inserting HTml tags into the label alongside the text including bold tags (****) to emphasize certain words, and a line break (**
**) to split the output over multiple lines. This is a common technique.

In Chapter 17, you'll learn how to get file and directory information and read and write files. For now, the simple application just displays the custom web.config setting, as shown in Figure 5-12.



Figure 5-12. Displaying custom application settings

ASP.NET web servers are configured, by default, to deny any requests for .config files. This means remote users will not be able to access the file. Instead, they'll receive the error message shown in Figure 5-13.

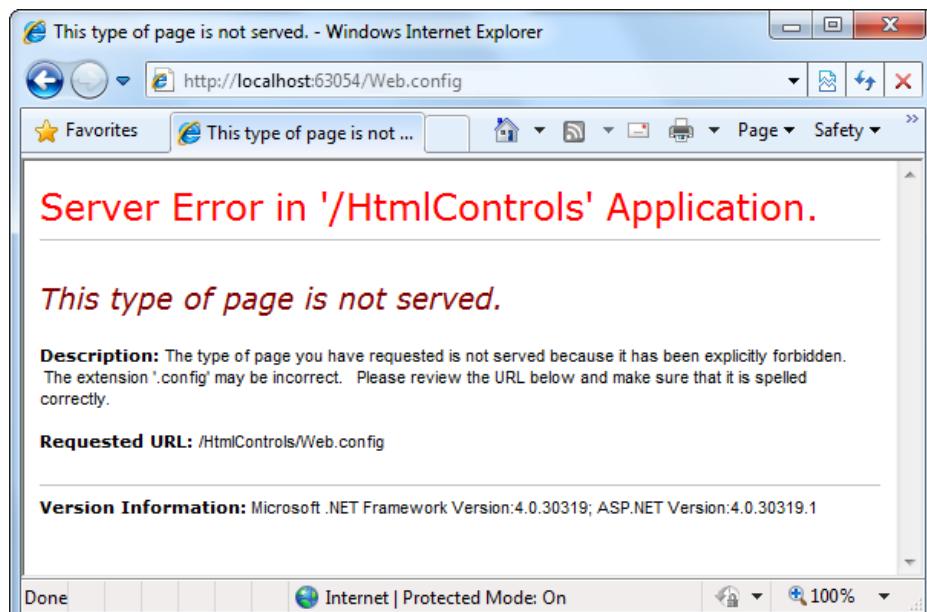


Figure 5-13. Requests for web.config are denied.

Using the Website Administration Tool

Editing the web.config file by hand is refreshingly straightforward, but it can be a bit tedious. To help alleviate the drudgery, ASP.NET includes a graphical configuration tool called the *Website Administration Tool (WAT)*, which lets you configure various parts of the web.config file by using a web page interface. To run the WAT to configure the current web project in Visual Studio, select Website à ASP.NET Configuration (or click the ASP.NET Configuration icon that's at the top of the Solution Explorer). A web browser window will appear (see Figure 5-14). Internet Explorer will automatically log you on under the current Windows user account, allowing you to make changes.

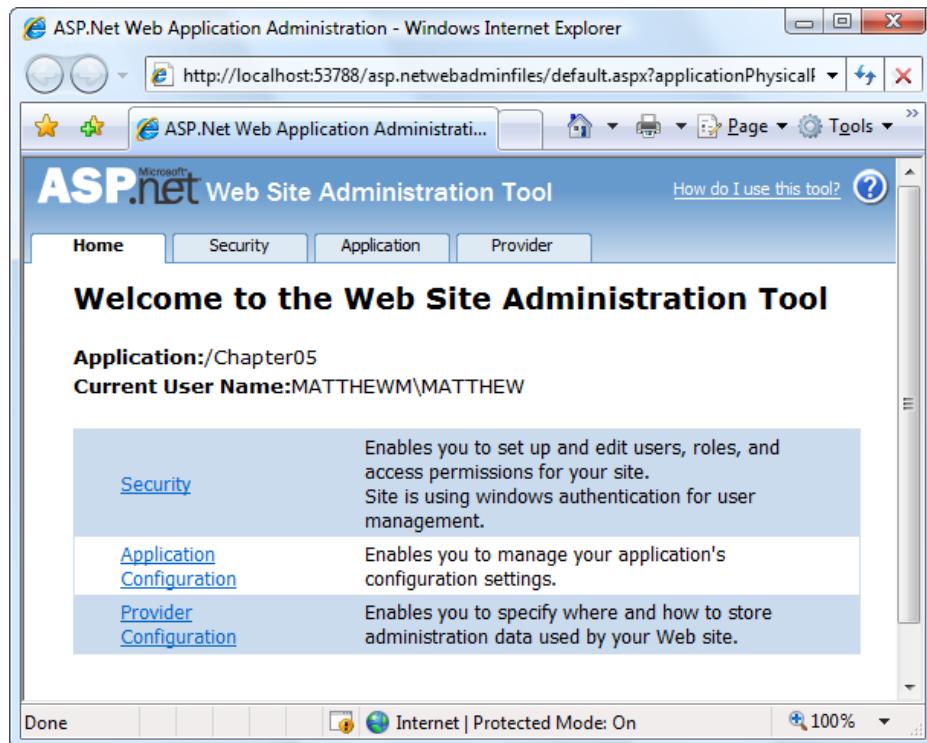


Figure 5-14. Running the WAT

You can use the WAT to automate the web.config changes you made in the previous example. To try this, click the Application tab. Using this tab, you can create a new setting (click the Create Application Settings link). If you click Manage Application Settings, you'll see a list with all the settings that are defined in your application (Figure 5-15). You can then choose to remove or edit any one of them.

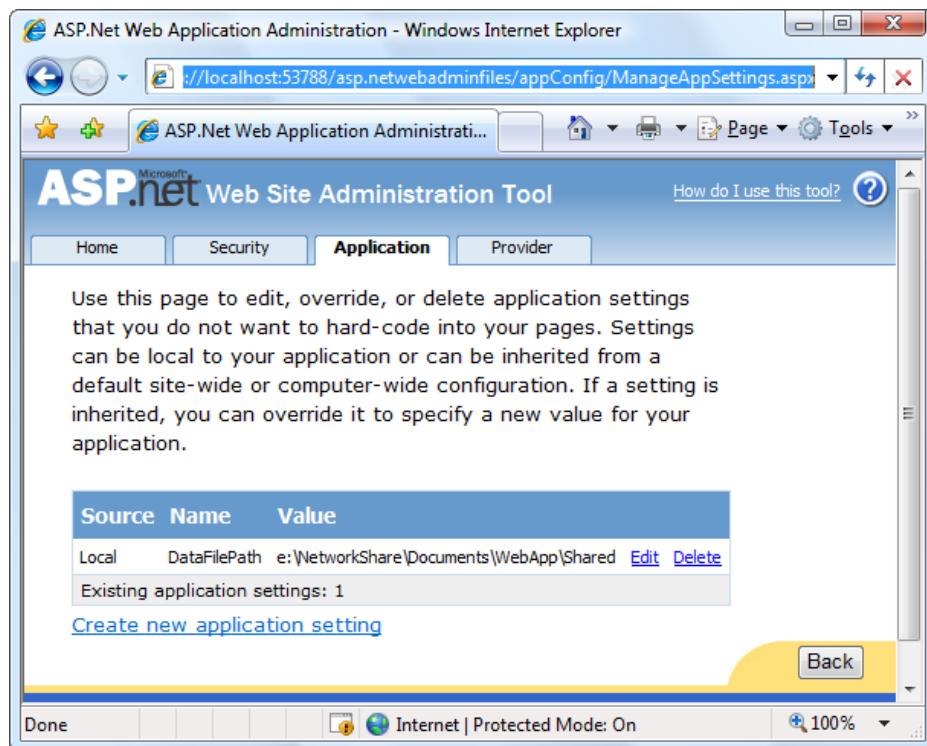


Figure 5-15. Editing an application setting with the WAT

This is the essential idea behind the WAT. You make your changes using a graphical interface (a web page), and the WAT generates the settings you need and adds them to the web.config file for your application behind the scenes. Of course, the WAT has a number of settings for configuring more-complex ASP.NET settings, and you'll use it throughout this book.

Note The WAT works only while you're developing a web application. In other words, you can't deploy to a live web server and then attempt to use the WAT. However, if you need to reconfigure an already-deployed application, you can use the graphical IIS manager tool, which provides some of the same features as the WAT (and additional ones). You'll learn more about IIS configuration in Chapter 26.

The Last Word

This chapter presented you with your first look at three core ASP.NET topics: web applications, server controls, and configuration. You should now understand how to create a very simple ASP.NET page, react to its events, and manipulate its content in code.

HTML controls are a compromise between ASP.NET web controls and traditional HTML. They use the familiar HTML elements but provide a limited object-oriented interface. Essentially, HTML controls are designed to be straightforward, predictable, and automatically compatible with existing programs. With HTML controls, the final HTML page that is sent to the client closely resembles the markup in the original .aspx page.

In the next chapter, you'll learn about web controls, which provide a more sophisticated object interface over the top of the underlying HTML. If you're starting a new project or need to add some of ASP.NET's most powerful controls, web controls are the best option.



Web Controls

The previous chapter introduced the event-driven and control-based programming model of ASP.NET. This model allows you to create programs for the Web by using the same object-oriented techniques you would use to write an ordinary desktop application.

However, HTML server controls really show only a glimpse of what is possible with ASP.NET's server control model. To see some of the real advantages, you need to use the richer and more extensible *web controls*. In this chapter, you'll explore the basic web controls and their class hierarchy. You'll also delve deeper into ASP.NET's event handling and learn the details of the web page life cycle. Finally, you'll put your knowledge to work by creating a web page that lets the user design a greeting card.

Stepping Up to Web Controls

Now that you've seen the new model of server controls, you might wonder why you need additional web controls. But in fact, HTML controls are much more limited than server controls need to be. For example, every HTML control corresponds directly to a single HTML element. Web controls, on the other hand, have no such restriction—they can switch from one element to another depending on how you're using them, or they can render themselves by using a complex combination of multiple elements.

These are some of the reasons you should switch to web controls:

They provide a rich user interface: A web control is programmed as an object but doesn't necessarily correspond to a single element in the final HTML page. For example, you might create a single Calendar or GridView control, which will be rendered as dozens of HTML elements in the final page. When using ASP.NET programs, you don't need to know the lower-level HTML details. The control creates the required HTML tags for you.

They provide a consistent object model: HTML is full of quirks and idiosyncrasies. For example, a simple text box can appear as one of three elements, including <textarea>, <input type="text">, and <input type="password">. With web controls, these three elements are consolidated as a single TextBox control. Depending on the properties you set, the underlying HTML element that ASP.NET renders may differ. Similarly, the names of properties don't follow the HTML attribute names. For example, controls that display text, whether it's a caption or a text box that can be edited by the user, expose a Text property.

They tailor their output automatically: ASP.NET server controls can detect the type of browser and automatically adjust the HTML code they write to take advantage of features such as support for JavaScript. You don't need to know about the client because ASP.NET handles that layer and automatically uses the best possible set of features. This feature is known as *adaptive rendering*.

They provide high-level features: You'll see that web controls allow you to access additional events, properties, and methods that don't correspond directly to typical HTML controls. ASP.NET implements these features by using a combination of tricks.

Throughout this book, you'll see examples that use the full set of web controls. To master ASP.NET development, you need to become comfortable with these user-interface ingredients and understand their abilities. HTML server controls, on the other hand, are less important for web forms development. You'll use them only if you're migrating an existing HTML page to the ASP.NET world, or if you need to have fine-grained control over the HTML code that will be generated and sent to the client.

Basic Web Control Classes

If you've ever created a Windows application before, you're probably familiar with the basic set of standard controls including labels, buttons, and text boxes. ASP.NET provides web controls for all these standbys. (And if you've created .NET Windows applications, you'll notice that the class names and properties have many striking similarities which are designed to make it easy to transfer the experience you acquire in one type of application to another.)

Table 6-1 lists the basic control classes and the HTML elements they generate. Some controls (such as Button and TextBox) can be rendered as different HTML elements. In this case, ASP.NET uses the element that matches the properties you've set. Also, some controls have no real HTML equivalent. For example, when the CheckBoxList and RadioButtonList controls render themselves, they may output a <table> that contains multiple HTML check boxes or radio buttons. ASP.NET exposes them as a single object on the server side for convenient programming, thus illustrating one of the primary strengths of web controls.

Table 6-1. Basic Web Controls

Control Class	Underlying HTML Element
Label	
Button	<input type="submit"> or <input type="button">
TextBox	<input type="text">, <input type="password">, or <textarea>
CheckBox	<input type="checkbox">
RadioButton	<input type="radio">
Hyperlink	<a>
LinkButton	<a> with a contained tag
ImageButton	<input type="image">
Image	
ListBox	<select size="X"> where X is the number of rows that are visible at once
DropDownList	<select>
CheckBoxList	A list or <table> with multiple <input type="checkbox"> tags
RadioButtonList	A list or <table> with multiple <input type="radio"> tags
BulletedList	An ordered list (numbered) or unordered list (bulleted)
Panel	<div>
Table, TableRow, and TableCell	<table>, <tr>, and <td> or <th>

This table omits some of the more specialized controls used for data, navigation, security, and web portals. You'll see these controls as you learn about their features throughout this book.

The Web Control Tags

ASP.NET tags have a special format. They always begin with the prefix `asp:` followed by the class name. If there is no closing tag, the tag must end with `/>`. (This syntax convention is borrowed from XML, which you'll learn about in much more detail in Chapter 18.) Each attribute in the tag corresponds to a control property, except for the `runat="server"` attribute, which signals that the control should be processed on the server.

The following, for example, is an ASP.NET TextBox:

```
<asp:TextBox ID="txt" runat="server" />
```

When a client requests this .aspx page, the following HTML is returned. The name is a special attribute that ASP.NET uses to track the control.

```
<input type="text" ID="txt" name="txt" />
```

Alternatively, you could place some text in the TextBox, set its size, make it read-only, and change the background color. All these actions have defined properties. For example, the `TextBox.TextMode` property allows you to specify `SingleLine` (the default), `MultiLine` (for a `<textarea>` type of control), or `Password` (for an input control that displays bullets to hide the true value). You can adjust the color by using the `BackColor` and `ForeColor` properties. And you can tweak the size of the TextBox in one of two ways—either using the `Rows` and `Columns` properties (for the pure HTML approach) or using the `Height` and `Width` properties (for the style-based approach). Both have the same result.

Here's one example of a customized TextBox:

```
<asp:TextBox ID="txt" BackColor="Yellow" Text="Hello World"  
ReadOnly="True" TextMode="MultiLine" Rows="5" runat="server" />
```

The resulting HTML uses the `<textarea>` element and sets all the required attributes (such as `rows` and `readonly`) and the `style` attribute (with the background color). It also sets the `cols` attribute with the default width of 20 columns, even though you didn't explicitly set the `TextBox.Columns` property:

```
<textarea name="txt" rows="5" cols="20" readonly="readonly" ID="txt"  
style="background-color:Yellow;">Hello World</textarea>
```

Figure 6-1 shows the `<textarea>` element in the browser.



Figure 6-1. A customized text box

Clearly, it's easy to create a web control tag. It doesn't require any understanding of HTML. However, you will need to understand the control class and the properties that are available to you.

Case-sensitivity in asp.net Forms

The .aspx layout portion of a web page tolerates different capitalization for tag names, property names, and enumeration values. For example, the following two tags are equivalent, and both will be interpreted by the ASP.NET engine, even though their case differs:

```
<asp:Button ID="Button1" runat="server"
    Enabled="False" Text="Button" Font-Size="XX-Small" />
<asp:button ID="Button2" runat="server"
    Enabled="false" tExT="Button" Font-Size="xx-small" />
```

This design was adopted to make .aspx pages behave more like ordinary HTML web pages, which ignore case completely. However, you can't use the same looseness in the tags that apply settings in the web.config file or the machine.config file. Here, case must match *exactly*.

Web Control Classes

Web control classes are defined in the System.Web.UI.WebControls namespace. They follow a slightly more tangled object hierarchy than HTML server controls. Figure 6-2 shows most, but not all, of the web controls that ASP.NET provides.

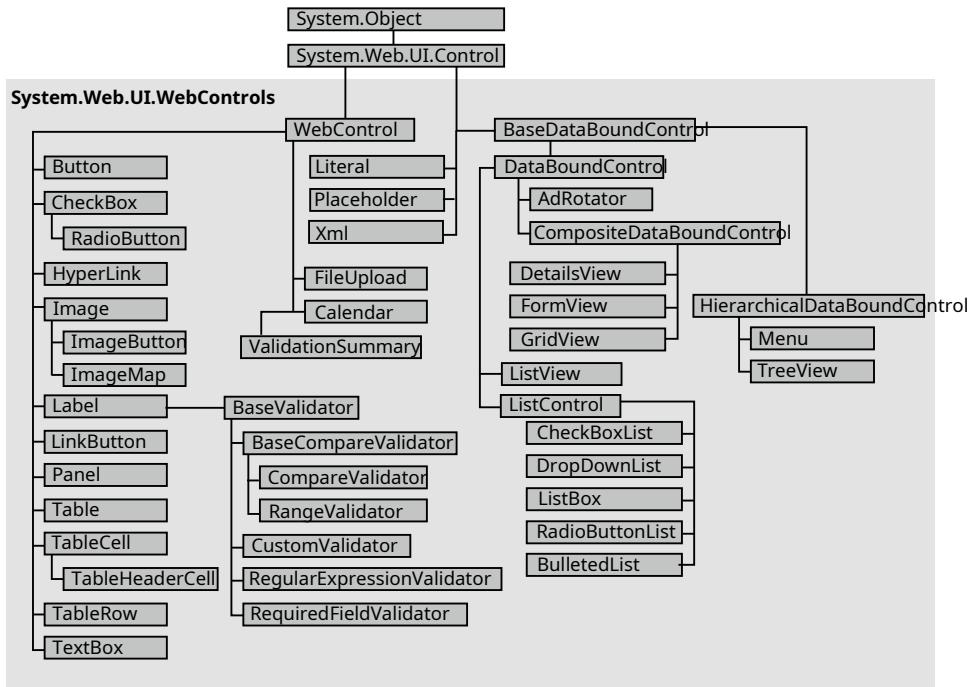


Figure 6-2. The web control hierarchy

This inheritance diagram includes some controls that you won't study in this chapter, including the data controls (such as the GridView, DetailsView, and FormView) and the validation controls. You'll explore these controls in later chapters.

The WebControl Base Class

Most web controls begin by inheriting from the WebControl base class. This class defines the essential functionality for tasks such as data binding and includes some basic properties that you can use with almost any web control, as described in Table 6-2.

Table 6-2. WebControl Properties

Property	Description
AccessKey	Specifies the keyboard shortcut as one letter. For example, if you set this to Y, the Alt+Y keyboard combination will automatically change focus to this web control (assuming the browser supports this feature).
BackColor, ForeColor, and BorderColor	Sets the colors used for the background, foreground, and border of the control. In most controls, the foreground color sets the text color.
BorderWidth	Specifies the size of the control border.
BorderStyle	One of the values from the BorderStyle enumeration, including Dashed, Dotted, Double, Groove, Ridge, Inset, Outset, Solid, and None.
Controls	Provides a collection of all the controls contained inside the current control. Each object is provided as a generic System.Web.UI.Control object, so you will need to cast the reference to access control-specific properties.
Enabled	When set to false, the control will be visible, but it will not be able to receive user input or focus.
EnableViewState	Set this to false to disable the automatic state management for this control. In this case, the control will be reset to the properties and formatting specified in the control tag (in the .aspx page) every time the page is posted back. If this is set to true (the default), the control uses the hidden input field to store information about its properties, ensuring that any changes you make in code are remembered.
Font	Specifies the font used to render any text in the control as a System.Web.UI.WebControls.FontInfo object.
Height and Width	Specifies the width and height of the control. For some controls, these properties will be ignored when used with older browsers.
ID	Specifies the name that you use to interact with the control in your code (and also serves as the basis for the ID that's used to name the top-level element in the rendered HTML).
Page	Provides a reference to the web page that contains this control as a System.Web.UI.Page object.
Parent	Provides a reference to the control that contains this control. If the control is placed directly on the page (rather than inside another control), it will return a reference to the page object.
TabIndex	A number that allows you to control the tab order. The control with a TabIndex of 0 has the focus when the page first loads. Pressing Tab moves the user to the control with the next lowestTabIndex, provided it is enabled. This property is supported only in Internet Explorer.
ToolTip	Displays a text message when the user hovers the mouse above the control. Many older browsers don't support this property.
Visible	When set to false, the control will be hidden and will not be rendered to the final HTML page that is sent to the client.

The next few sections describe some of the common concepts you'll use with almost any web control, including how to set properties that use units and enumerations and how to use colors and fonts.

Units

All the properties that use measurements, including `BorderWidth`, `Height`, and `Width`, require the `Unit` structure, which combines a numeric value with a type of measurement (pixels, percentage, and so on). This means when you set these properties in a control tag, you must make sure to append `px` (pixel) or `%` (for percentage) to the number to indicate the type of unit.

Here's an example with a `Panel` control that is 300 pixels high and has a width equal to 50 percent of the current browser window:

```
<asp:Panel Height="300px" Width="50%" ID="pnl" runat="server" />
```

If you're assigning a unit-based property through code, you need to use one of the static methods of the `Unit` type. Use `Pixel()` to supply a value in pixels, and use `Percentage()` to supply a percentage value:

```
// Convert the number 300 to a Unit object
// representing pixels, and assign it.
pnl.Height = Unit.Pixel(300);
```

```
// Convert the number 50 to a Unit object
// representing percent, and assign it.
pnl.Width = Unit.Percentage(50);
```

You could also manually create a `Unit` object and initialize it by using one of the supplied constructors and the `UnitType` enumeration. This requires a few more steps but allows you to easily assign the same unit to several controls:

```
// Create a Unit object.
Unit myUnit = new Unit(300, UnitType.Pixel);

// Assign the Unit object to several controls or properties.
pnl.Height = myUnit;
pnl.Width = myUnit;
```

Enumerations

Enumerations are used heavily in the .NET class library to group a set of related constants. For example, when you set a control's `BorderStyle` property, you can choose one of several predefined values from the `BorderStyle` enumeration. In code, you set an enumeration by using the dot syntax:

```
ctrl.BorderStyle = BorderStyle.Dashed;
```

In the `.aspx` file, you set an enumeration by specifying one of the allowed values as a string. You don't include the name of the enumeration type, which is assumed automatically.

```
<asp:Label BorderStyle="Dashed" Text="Border Test" ID="lbl"
runat="server" />
```

Figure 6-3 shows the label with the altered border.

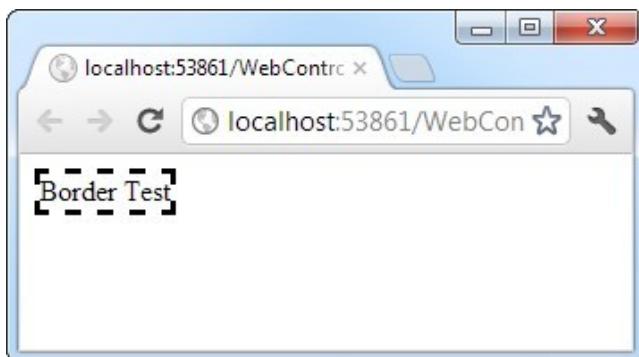


Figure 6-3. Modifying the border style

Colors

The Color property refers to a Color object from the System.Drawing namespace. You can create color objects in several ways:

Using an ARGB (alpha, red, green, blue) color value: You specify each value as an integer from 0 to 255. The alpha component represents the transparency of a color, and usually you'll use 255 to make the color completely opaque.

Using a predefined .NET color name: You choose the correspondingly named read-only property from the Color structure. These properties include the 140 HTML color names.

Using an HTML color name: You specify this value as a string by using the ColorTranslator class.

To use any of these techniques, you'll probably want to start by importing the System.Drawing namespace, as follows:

```
using System.Drawing;
```

The following code shows several ways to specify a color in code:

```
// Create a color from an ARGB value
int alpha = 255, red = 0, green = 255, blue = 0;
ctrl.ForeColor = Color.FromArgb(alpha, red, green, blue);

// Create a color using a .NET name
ctrl.ForeColor = Color.Crimson;

// Create a color from an HTML code
ctrl.ForeColor = ColorTranslator.FromHtml("Blue");
```

When defining a color in the .aspx file, you can use any one of the known color names:

```
<asp:TextBox ForeColor="Red" Text="Test" ID="txt" runat="server" />
```

The HTML color names that you can use are listed in the MSDN reference website (see <http://msdn.microsoft.com/library/system.drawing.color.aspx>). Alternatively, you can use a hexadecimal color number (in the format # <red> <green> <blue>), as shown here:

```
<asp:TextBox ForeColor="#ff50ff" Text="Test"
    ID="txt" runat="server" />
```

Fonts

The Font property actually references a full `FontInfo` object, which is defined in the `System.Web.UI.WebControls` namespace. Every `FontInfo` object has several properties that define its name, size, and style (see Table 6-3).

Table 6-3. *FontInfo Properties*

Property	Description
Name	A string indicating the font name (such as Verdana).
Names	An array of strings with font names, in the order of preference. The browser will use the first matching font that's installed on the user's computer.
Size	The size of the font as a <code>FontUnit</code> object. This can represent an absolute or relative size.
Bold, Italic, Strikeout, Underline, and Overline Boolean properties that apply the given style attribute.	

In code, you can assign a font by using the familiar dot syntax to set the various font properties:

```
ctrl.Font.Name = "Verdana";
ctrl.Font.Bold = true;
```

You can also set the size by using the `FontUnit` type:

```
// Specifies a relative size.
ctrl.Font.Size = FontUnit.Small;
```

```
// Specifies an absolute size of 14 pixels.
ctrl.Font.Size = FontUnit.Point(14);
```

In the .aspx file, you need to use a special “object walker” syntax to specify object properties such as `Font`. The object walker syntax uses a hyphen (-) to separate properties. For example, you could set a control with a specific font (Tahoma) and font size (40 point) like this:

```
<asp:TextBox Font-Name="Tahoma" Font-Size="40" Text="Size Test" ID="txt"
    runat="server" />
```

Or you could set a relative size like this:

```
<asp:TextBox Font-Name="Tahoma" Font-Size="Large" Text="Size Test"
    ID="txt" runat="server" />
```

Figure 6-4 shows the altered TextBox in this example.

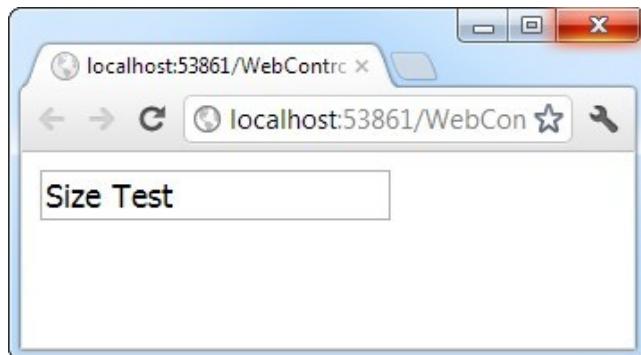


Figure 6-4. Modifying a control's font

A font setting is really just a recommendation. If the client computer doesn't have the font you request, it reverts to a standard font. To deal with this problem, it's common to specify a list of fonts, in order of preference. To do so, you use the `Font.Names` property instead of `Font.Name`, as shown here:

```
<asp:TextBox Font-Names="Verdana,Tahoma,Arial"  
Text="Size Test" ID="txt" runat="server" />
```

Here, the browser will use the Verdana font (if it has it). If not, it will fall back on Tahoma or, if that isn't present, Arial.

When specifying fonts, it's a good idea to end with one of the following fonts, which are supported on all browsers:

- Times
- Arial and Helvetica
- Courier

The following fonts are found on almost all Windows and Mac computers, but not necessarily on other operating systems:

- Verdana
- Georgia
- Tahoma
- Comic Sans
- Arial Black
- Impact

embedded Fonts

Recently, browser support has been growing for a feature called *embedded fonts* or *web fonts*. The basic idea is to allow you to format text in your web page with a fancy, nonstandard web font. When someone views the page, the browser downloads the corresponding font file automatically, and uses it (temporarily) to display the text in the page. The embedded font feature allows your page to use a virtually unlimited number of typefaces, without worrying about what clients have installed on their computers.

because embedded fonts aren't universally supported (particularly in older browsers), they're generally used as an optional way to *enhance* the appearance of headings. That way, the web page can always fall back on one of the standard fonts described in the previous section if the web font doesn't work or it can't be downloaded. Usually, web designers don't use embedded fonts to format entire paragraphs of text, because of the risk that the text won't display properly on some browsers.

Technically, embedded fonts are a feature of CSS3. There is no ASP.NET-specific way to use embedded fonts. If you decide to use embedded fonts in an ASP.NET page, you will probably use them to format an ordinary HTML element, like a `<p>` paragraph, a ``, or a numbered heading (`<h1>`, `<h2>`, `<h3>`, and so on). You won't typically use them to format a web control (although you could, by setting the control's `CssClass` property to the CSS style that applies the embedded font). but in any case, it's up to you to add the styling settings to your style sheet.

Using embedded fonts is a somewhat awkward procedure with several steps. First, you need the right files, which you must upload to your web server. The challenge is that different browsers support different web font formats, with no single universal standard, so you'll need to include multiple copies of the same font. Furthermore, before you begin converting and uploading a font file you own, you need to make sure it's licensed for web use (most aren't). You can get a quick summary [of the pros and cons](http://www.phpdesk.com/en/tutorials/webfonts/quick) . but by far, the easiest strategy is to simply use a web font service such as Google Web Fonts. There you can pick from a huge gallery of slick web fonts. When you choose a font, Google will generate the corresponding CSS style—all you need to do is just paste it into the appropriate part of your style sheet. Google also makes all its web fonts available through its own web servers, so there are no extra files to upload to your website. To learn more, visit www.google.com/webfonts

Focus

Unlike HTML server controls, every web control provides a `Focus()` method. The `Focus()` method affects only input controls (controls that can accept keystrokes from the user). When the page is rendered in the client browser, the user starts in the focused control.

For example, if you have a form that allows the user to edit customer information, you might call the `Focus()` method on the first text box in that form. That way, the cursor appears in this text box immediately when the page first loads in the browser. If the text box is partway down the form, the page even scrolls down to it automatically. The user can then move from control to control by using the time-honored Tab key.

If you're a seasoned HTML developer, you know there isn't any built-in way to give focus to an input control. Instead, you need to rely on JavaScript. This is the secret to ASP.NET's implementation. When your code is finished processing and the page is rendered, ASP.NET adds an extra block of JavaScript code to the end of your page. This JavaScript code simply sets the focus to the last control that used the `Focus()` method. If you haven't called `Focus()` at all, this code isn't added to the page.

Rather than call the `Focus()` method programmatically, you can set a control that should always be focused by setting the `DefaultFocus` property of the `<form>` tag:

```
<form DefaultFocus="TextBox2" runat="server">
```

You can override the default focus by calling the Focus() method in your code.

Another way to manage focus is by using access keys. For example, if you set the AccessKey property of a TextBox to A, pressing Alt+A will switch focus to the TextBox. Labels can also get into the game, even though they can't accept focus. The trick is to set the Label.AssociatedControlID property to specify a linked input control. That way, the label transfers focus to the associated control.

For example, the following label gives focus to TextBox2 when the keyboard combination Alt+2 is pressed:

```
<asp:Label AccessKey="2" AssociatedControlID="TextBox2" runat="server"
Text="TextBox2:" />
<asp:TextBox runat="server" ID="TextBox2" />
```

Focusing and access keys are also supported in non-Microsoft browsers, including Firefox.

The Default Button

Along with control focusing, ASP.NET also allows you to designate a default button on a web page. The default button is the button that is "clicked" when the user presses the Enter key. For example, if your web page includes a form, you might want to make the submit button into a default button. That way, if the user hits Enter at any time, the page is posted back and the Button.Click event is fired for that button.

To designate a default button, you must set the HtmlForm.DefaultButton property with the ID of the respective control, as shown here:

```
<form DefaultButton="cmdSubmit" runat="server">
```

The default button must be a control that implements the IButtonControl interface. The interface is implemented by the Button, LinkButton, and ImageButton web controls but not by any of the HTML server controls.

In some cases, it makes sense to have more than one default button. For example, you might create a web page with two groups of input controls. Both groups may need a different default button. You can handle this by placing the groups into separate panels. The Panel control also exposes the DefaultButton property, which works when any input control it contains gets the focus.

Control prefixes

When working with web controls, it's often useful to use a three-letter lowercase prefix to identify the type of control. The preceding example (and those in the rest of this book) follows this convention to make the interface code as clear as possible. Some recommended control prefixes are as follows:

- button: cmd (or btn)
- Checkbox: chk
- Image: img
- label: lbl
- list control: lst
- Panel: pnl
- Radiobutton: opt
- Textbox: txt

If you're a veteran programmer, you'll also notice that this book doesn't use prefixes to identify data types. This is in keeping with the philosophy of .NET, which recognizes that data types can often change freely without consequence and that variables often point to full-featured objects instead of simple data values.

List Controls

The list controls include the `ListBox`, `DropDownList`, `CheckBoxList`, `RadioButtonList`, and `BulletedList`. They all work in essentially the same way but are rendered differently in the browser. The `ListBox`, for example, is a rectangular list that displays several entries, while the `DropDownList` shows only the selected item. The `CheckBoxList` and `RadioButtonList` are similar to the `ListBox`, but every item is rendered as a check box or option button, respectively. Finally, the `BulletedList` is the odd one out—it's the only list control that isn't selectable. Instead, it renders itself as a sequence of numbered or bulleted items.

All the selectable list controls provide a `SelectedIndex` property that indicates the selected row as a zero-based index (just like the `HtmlSelect` control you used in the previous chapter). For example, if the first item in the list is selected, the `SelectedIndex` will be 0. Selectable list controls also provide an additional `SelectedItem` property, which allows your code to retrieve the `ListItem` object that represents the selected item. The `ListItem` object provides three important properties: `Text` (the displayed content), `Value` (the hidden value from the HTML markup), and `Selected` (true or false depending on whether the item is selected).

In the previous chapter, you used code like this to retrieve the selected `ListItem` object from an `HtmlSelect` control called `Currency`, as follows:

```
ListItem item;  
item = Currency.Items[Currency.SelectedIndex];
```

If you used the `ListBox` web control, you can simplify this code with a clearer syntax:

```
ListItem item;  
item = Currency.SelectedItem;
```

Multiple-Select List Controls

Some list controls can allow multiple selections. This isn't allowed for the `DropDownList` or `RadioButtonList`, but it is supported for a `ListBox`, provided you have set the `SelectionMode` property to the enumerated value `ListSelectionMode.Multiple`. The user can then select multiple items by holding down the `Ctrl` key while clicking the items in the list. With the `CheckBoxList`, multiple selections are always possible.

The `SelectedIndex` and `SelectedItem` properties aren't much help with a list that supports multiple selection, because they will simply return the first selected item. Instead, you can find all the selected items by iterating through the `Items` collection of the list control and checking the `ListItem.Selected` property of each item. (If it's true, that item is one of the selected items.) Figure 6-5 shows a simple web page example that provides a list of computer languages. After the user clicks the `OK` button, the page indicates which selections the user made.

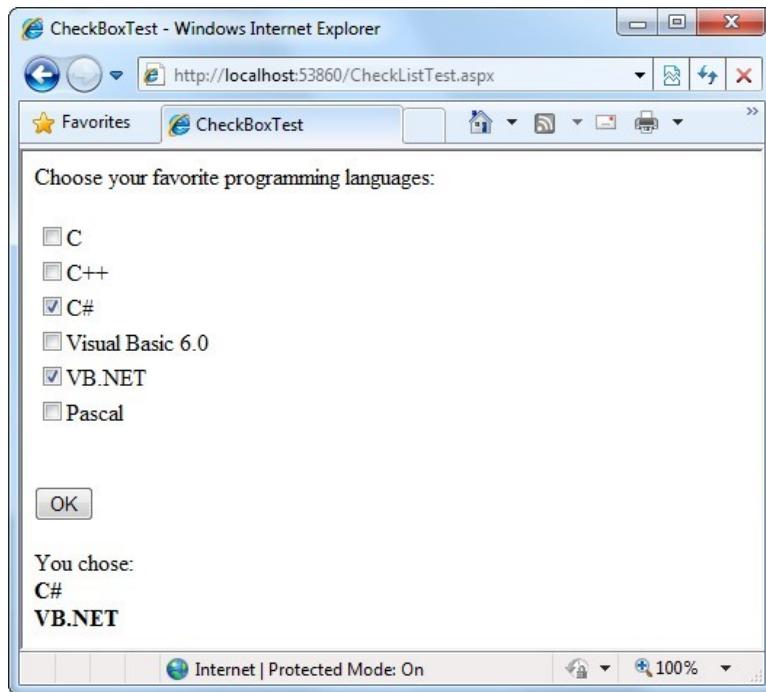


Figure 6-5. A simple CheckListBox test

The .aspx file for this page defines CheckListBox, Button, and Label controls, as shown here:

```
<%@ Page Language="C#" AutoEventWireup="true"
   CodeFile="CheckListTest.aspx.cs" Inherits="CheckListTest" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <title>CheckBoxTest</title>
</head>
<body>
  <form runat="server">
    <div>
      Choose your favorite programming languages:<br /><br />
      <asp:CheckBoxList ID="chklst" runat="server" /><br /><br />
      <asp:Button ID="cmdOK" Text="OK" OnClick="cmdOK_Click" runat="server" />
      <br /><br />
      <asp:Label ID="lblResult" runat="server" />
    </div>
  </form>
</body>
</html>
```

The code adds items to the CheckBoxList at startup and iterates through the collection when the button is clicked:

```
public partial class CheckListTest : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            chklst.Items.Add("C");
            chklst.Items.Add("C++");
            chklst.Items.Add("C#");
            chklst.Items.Add("Visual Basic 6.0");
            chklst.Items.Add("VB.NET");
            chklst.Items.Add("Pascal");
        }
    }

    protected void cmdOK_Click(object sender, EventArgs e)
    {
        lblResult.Text = "You chose:<b>";

        foreach (ListItem lstItem in chklst.Items)
        {
            if (lstItem.Selected == true)
            {
                // Add text to label.
                lblResult.Text += "<br />" + lstItem.Text;
            }
        }
        lblResult.Text += "</b>";
    }
}
```

The BulletedList Control

The BulletedList control is a server-side equivalent of the (unordered list) and (ordered list) elements. As with all list controls, you set the collection of items that should be displayed through the Items property. Additionally, you can use the properties in Table 6-4 to configure how the items are displayed.

Table 6-4. Added BulletedList Properties

Property	Description
BulletStyle	Determines the type of list. Choose from Numbered (1, 2, 3, . . .); LowerAlpha (a, b, c, . . .) and UpperAlpha (A, B, C, . . .); LowerRoman (i, ii, iii, . . .) and UpperRoman (I, II, III, . . .); and the bullet symbols Disc, Circle, Square, or CustomImage (in which case you must set the BulletImageUrl property).
BulletImageUrl	If the BulletStyle is set to CustomImage, this points to the image that is placed to the left of each item as a bullet.
FirstBulletNumber	In an ordered list (using the Numbered, LowerAlpha, UpperAlpha, LowerRoman, and UpperRoman styles), this sets the first value. For example, if you set FirstBulletNumber to 3, the list might read 3, 4, 5 (for Numbered) or C, D, E (for UpperAlpha).
DisplayMode	Determines whether the text of each item is rendered as text (use Text, the default) or a hyperlink (use LinkButton or HyperLink). The difference between LinkButton and HyperLink is how they treat clicks. When you use LinkButton, the BulletedList fires a Click event that you can react to on the server to perform the navigation. When you use HyperLink, the BulletedList doesn't fire the Click event—instead, it treats the text of each list item as a relative or absolute URL, and renders them as ordinary HTML hyperlinks. When the user clicks an item, the browser attempts to navigate to that URL.

If you set the DisplayMode to LinkButton, you can react to the Button.Click event to determine which item was clicked. For example, say you have a BulletedList like this:

```
<asp:BulletedList BulletStyle="Numbered" DisplayMode="LinkButton"
  ID="BulletedList1" OnClick="BulletedList1_Click" runat="server">
</asp:BulletedList>
```

You can use the following code to intercept its clicks:

```
protected void BulletedList1_Click(object sender, BulletedListEventArgs e)
{
    // Find the clicked item in the list.
    // (You can't use the SelectedIndex property here because static lists
    // don't support selection.)
    string itemText = BulletedList1.Items[e.Index].Text;
    Label1.Text = "You choose item" + itemText;
}
```

Figure 6-6 shows all the BulletStyle values that the BulletedList supports. When you click one of the items, the list changes to use that BulletStyle. You can try this example page with the sample WebControls project for this chapter.

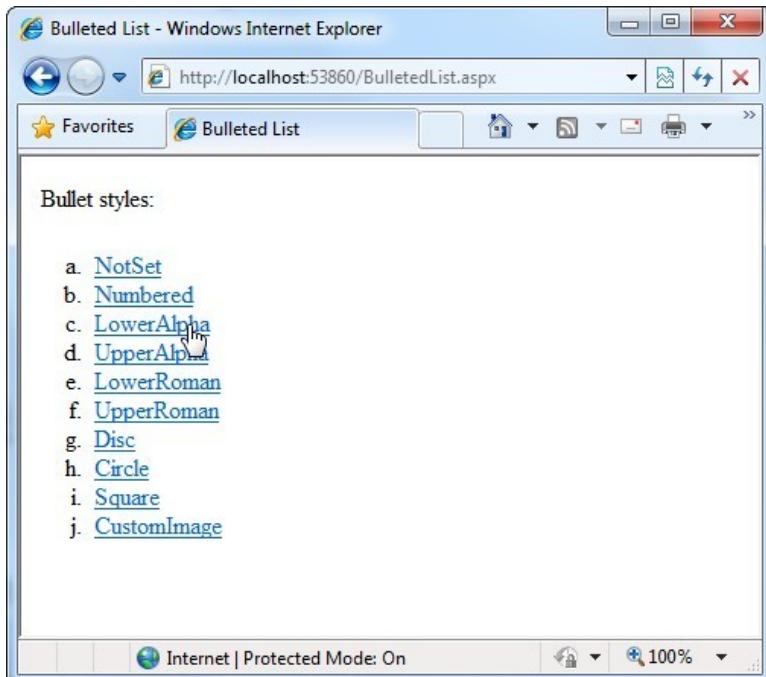


Figure 6-6. Various *BulletedList* styles

Table Controls

Essentially, the Table control is built out of a hierarchy of objects. Each Table object contains one or more TableRow objects. In turn, each TableRow object contains one or more TableCell objects. Each TableCell object contains other ASP.NET controls or HTML content that displays information. If you're familiar with the HTML table tags, this relationship (shown in Figure 6-7) will seem fairly logical.

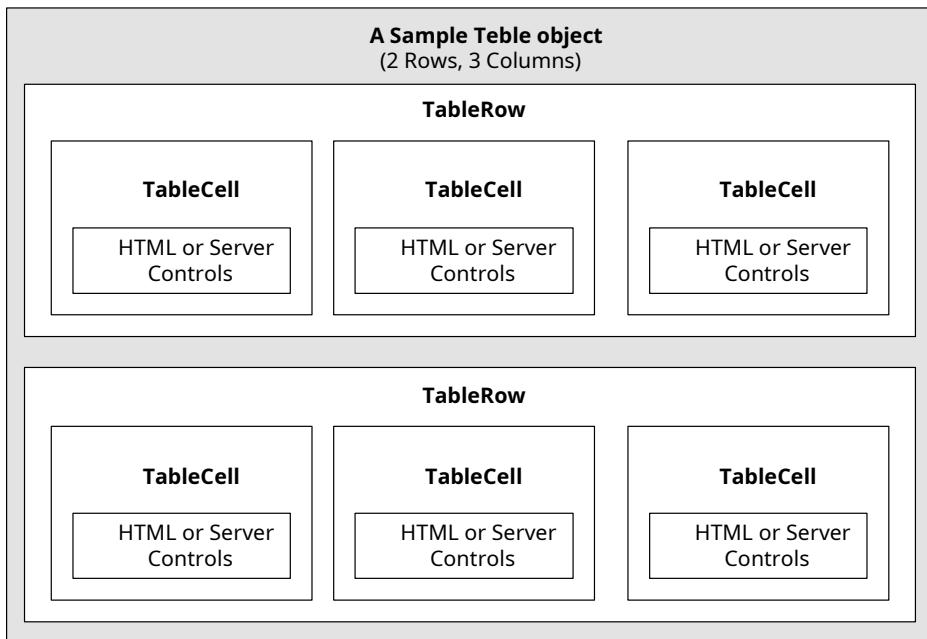


Figure 6-7. Table control containment

To create a table dynamically, you follow the same philosophy as you would for any other web control. First you create and configure the necessary ASP.NET objects. Then ASP.NET converts these objects to their final HTML representation before the page is sent to the client.

Consider the example shown in Figure 6-8. It allows the user to specify a number of rows and columns as well as whether cells should have borders.

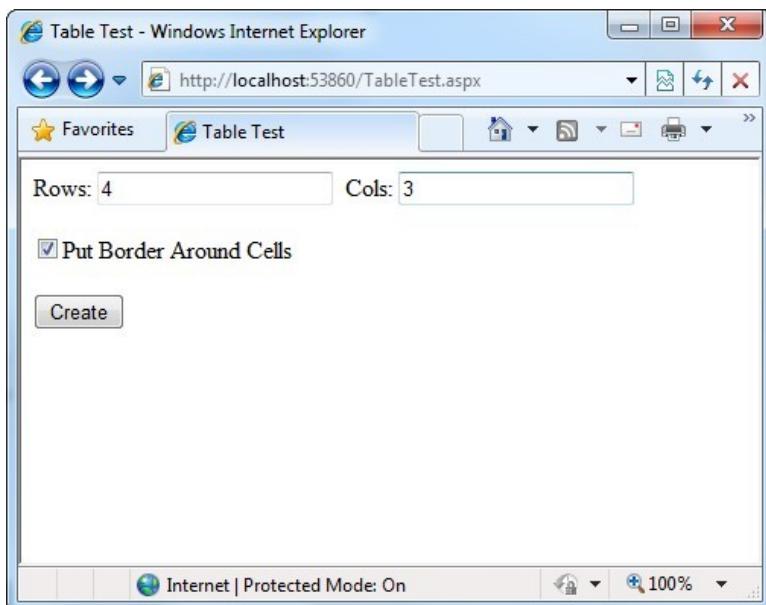


Figure 6-8. The table test options

When the user clicks the Create button, the table is filled dynamically with sample data according to the selected options, as shown in Figure 6-9.

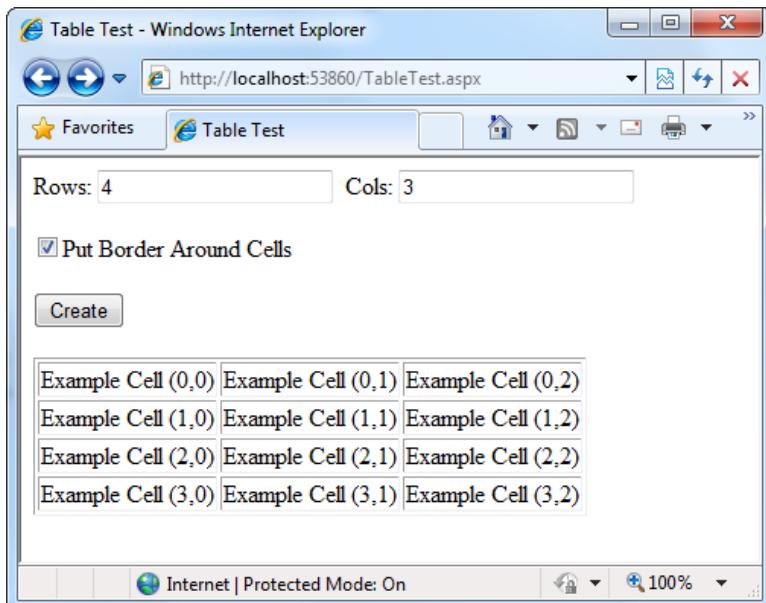


Figure 6-9. A dynamically generated table

The .aspx code creates the TextBox, CheckBox, Button, and Table controls:

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeFile="TableTest.aspx.cs" Inherits="TableTest" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <title>Table Test</title>
</head>
<body>
  <form runat="server">
    <div>
      Rows:
      <asp:TextBox ID="txtRows" runat="server" />
      &nbsp;Cols:
      <asp:TextBox ID="txtCols" runat="server" />
      <br /><br />
      <asp:CheckBox ID="chkBorder" runat="server"
        Text="Put Border Around Cells" />
      <br /><br />
      <asp:Button ID="cmdCreate" OnClick="cmdCreate_Click" runat="server"
        Text="Create" />
      <br /><br />
      <asp:Table ID="tbl" runat="server" />
    </div>
  </form>
</body>
</html>
```

You'll notice that the Table control doesn't contain any actual rows or cells. To make a valid table, you would need to nest several layers of tags. The following example creates a table with a single cell that contains the text *A Test Row*:

```
<asp:Table ID="tbl" runat="server">
  <asp:TableRow ID="row" runat="server">
    <asp:TableCell ID="cell" runat="server">A Sample Value</asp:TableCell>
  </asp:TableRow>
</asp:Table>
```

The table test web page doesn't have any nested elements. This means the table will be created as a server-side control object, but unless the code adds rows and cells, the table will not be rendered in the final HTML page.

The TableTest class uses two event handlers. When the page is first loaded, it adds a border around the table. When the button is clicked, it dynamically creates the required TableRow and TableCell objects in a loop.

```
public partial class TableTest : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    // Configure the table's appearance.
    // This could also be performed in the .aspx file
    // or in the cmdCreate_Click event handler.
    tbl.BorderStyle = BorderStyle.Inset;
    tbl.BorderWidth = Unit.Pixel(1);
  }
}
```

```

protected void cmdCreate_Click(object sender, EventArgs e)
{
    // Remove all the current rows and cells.
    // This is not necessary if EnableViewState is set to false.
    tbl.Controls.Clear();

    int rows = Int32.Parse(txtRows.Text);
    int cols = Int32.Parse(txtCols.Text);

    for (int row = 0; row < rows; row++)
    {
        // Create a new TableRow object.
        TableRow rowNew = new TableRow();

        // Put the TableRow in the Table.
        tbl.Controls.Add(rowNew);

        for (int col = 0; col < cols; col++)
        {
            // Create a new TableCell object.
            TableCell cellNew = new TableCell();

            cellNew.Text = "Example Cell (" + row.ToString() + ",";
            cellNew.Text += col.ToString() + ")";

            if (chkBorder.Checked)
            {
                cellNew.BorderStyle = BorderStyle.Inset;
                cellNew.BorderWidth = Unit.Pixel(1);
            }

            // Put the TableCell in the TableRow.
            rowNew.Controls.Add(cellNew);
        }
    }
}
}

```

This code uses the `Controls` collection to add child controls. Every container control provides this property. You could also use the `TableCell.Controls` collection to add web controls to each `TableCell`. For example, you could place an `Image` control and a `Label` control in each cell. In this case, you can't set the `TableCell.Text` property. The following code snippet uses this technique, and Figure 6-10 displays the results:

```

// Create a new TableCell object.
cellNew = new TableCell();

// Create a new Label object.
Label lblNew = new Label();
lblNew.Text = "Example Cell (" + row.ToString() + ",";
lblNew.Text += col.ToString() + "<br />";

```

```

System.Web.UI.WebControls.Image imgNew = new System.Web.UI.WebControls.Image();
imgNew.ImageUrl = "cellpic.png";

// Put the label and picture in the cell.
cellNew.Controls.Add(lblNew);
cellNew.Controls.Add(imgNew);

// Put the TableCell in the TableRow.
rowNew.Controls.Add(cellNew);

```

The real flexibility of the table test page is that each Table, TableRow, and TableCell is a full-featured object. If you want, you can give each cell a different border style, border color, and text color by setting the corresponding properties.

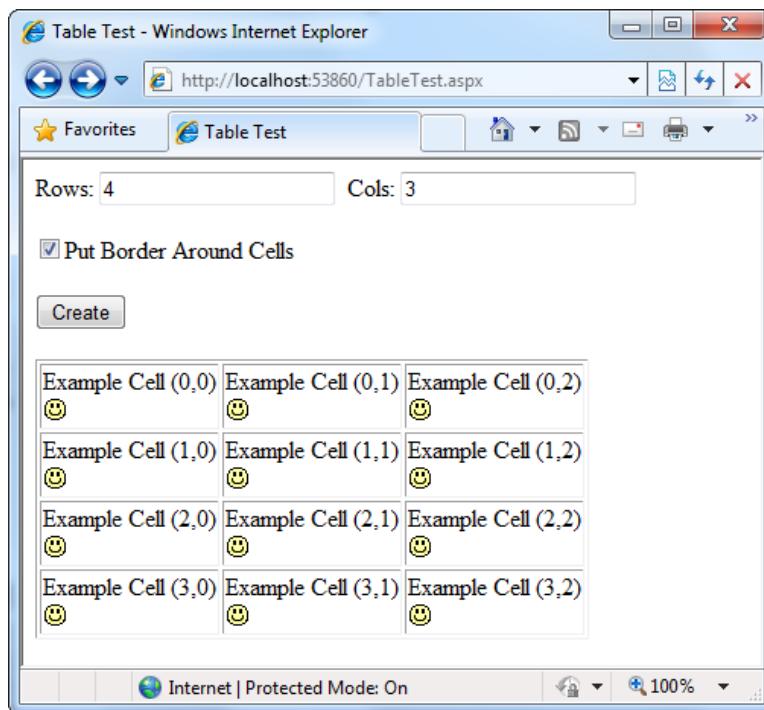


Figure 6-10. A table with contained controls

Web Control Events and AutoPostBack

The previous chapter explained that one of the main limitations of HTML server controls is their limited set of useful events—they have exactly two. HTML controls that trigger a postback, such as buttons, raise a ServerClick event. Input controls provide a ServerChange event that doesn't actually fire until the page is posted back.

ASP.NET server controls are really an ingenious illusion. You'll recall that the code in an ASP.NET page is processed on the server. It's then sent to the user as ordinary HTML. Figure 6-11 illustrates the order of events in page processing.

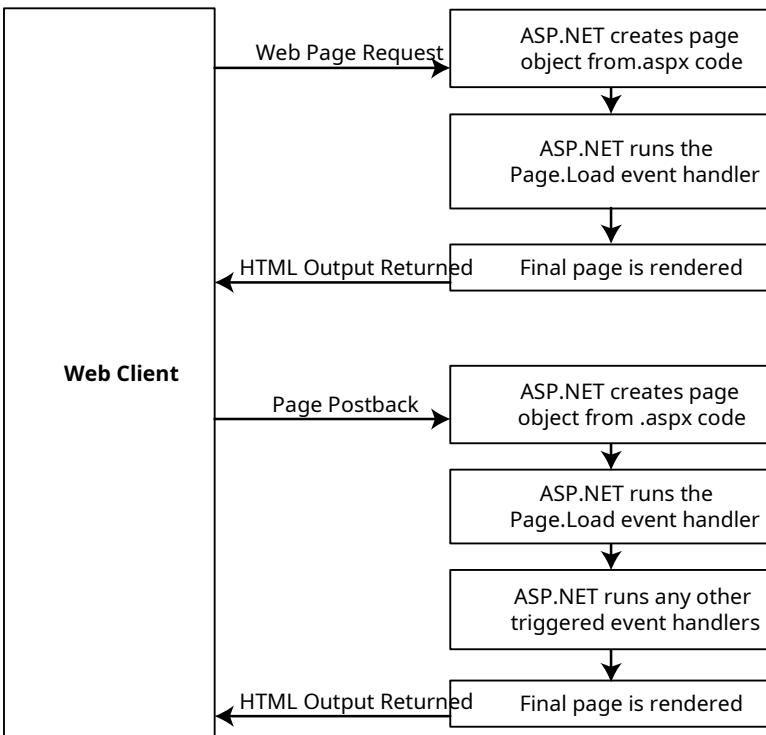


Figure 6-11. The page-processing sequence

This is the same in ASP.NET as it was in traditional ASP programming. The question is, how can you write server code that will react *immediately* to an event that occurs on the client?

Some events, such as the Click event of a button, do occur immediately. That's because when clicked, the button posts back the page. This is a basic convention of HTML forms. However, other actions *do* cause events but *don't* trigger a postback—for example, when the user changes the text in a text box (which triggers the TextChanged event) or chooses a new item in a list (the SelectedIndexChanged event). You might want to respond to these events, but without a postback, your code has no way to run.

ASP.NET handles this by giving you two options:

- You can wait until the next postback to react to the event. For example, imagine you want to react to the SelectedIndexChanged event in a list. If the user selects an item in a list, nothing happens immediately. However, if the user then clicks a button to post back the page, *two* events fire: Button.Click followed by ListBox.SelectedIndexChanged. And if you have several controls, it's quite possible for a single postback to result in several change events, which fire one after the other, in an undetermined order.
- You can use the *automatic postback* feature to force a control to post back the page immediately when it detects a specific user action. In this scenario, when the user clicks a new item in the list, the page is posted back, your code executes, and a new version of the page is returned.

The option you choose depends on the result you want. If you need to react immediately (for example, you want to update another control when a specific action takes place), you need to use automatic postbacks. On the other hand, automatic postbacks can sometimes make the page less responsive, because each postback and page refresh adds a short, but noticeable, delay and page refresh. (You'll learn how to create pages that update themselves without a noticeable page refresh when you consider ASP.NET AJAX in Chapter 25.)

All input web controls support automatic postbacks. Table 6-5 provides a basic list of web controls and their events.

Table 6-5. Web Control Events

Event	Web Controls That Provide It	Always Posts Back
Click	Button, ImageButton	True
TextChanged	TextBox (fires only after the user changes the focus to another control)	False
CheckedChanged	CheckBox, RadioButton	False
SelectedIndexChanged	DropDownList, ListBox, CheckBoxList, RadioButtonList	False

If you want to capture a change event (such as TextChanged, CheckedChanged, or SelectedIndexChanged) immediately, you need to set the control's AutoPostBack property to true. This way, the page will be submitted automatically when the user interacts with the control (for example, picks a selection in the list, clicks a radio button or a check box, or changes the text in a text box and then moves to a new control).

When the page is posted back, ASP.NET will examine the page, load all the current information, and then allow your code to perform some extra processing before returning the page back to the user (see Figure 6-12). Depending on the result you want, you could have a page that has some controls that post back automatically and others that don't.

This postback system isn't ideal for all events. For example, some events that you may be familiar with from Windows programs, such as mouse movement events or key press events, aren't practical in an ASP.NET application. Resubmitting the page every time a key is pressed or the mouse is moved would make the application unbearably slow and unresponsive.

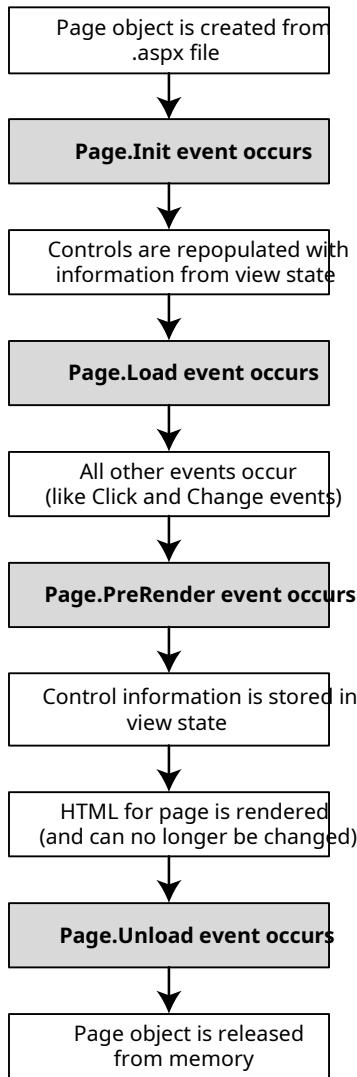


Figure 6-12. The postback processing sequence

How Postback Events Work

Chapter 1 explained that not all types of web programming use server-side code like ASP.NET. One common example of client-side web programming is JavaScript, which uses script code that's executed by the browser. ASP.NET uses the client-side abilities of JavaScript to bridge the gap between client-side and server-side code.

(Another scripting language is VBScript, but JavaScript is the only one that works on all modern browsers, including Internet Explorer, Chrome, Firefox, Safari, and Opera.)

Here's how it works: If you create a web page that includes one or more web controls that are configured to use AutoPostBack, ASP.NET adds a special JavaScript function to the rendered HTML page. This function is named `_doPostBack()`. When called, it triggers a postback, sending data back to the web server.

ASP.NET also adds two hidden input fields that are used to pass information back to the server. This information consists of the ID of the control that raised the event and any additional information that might be relevant. These fields are initially empty, as shown here:

```
<input type="hidden" name="__EVENTTARGET" ID="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" ID="__EVENTARGUMENT" value="" />
```

The `_doPostBack()` function has the responsibility of setting these values with the appropriate information about the event and then submitting the form. The `_doPostBack()` function is shown here:

```
<script language="text/javascript">
function _doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
    ...
}
</script>
```

Remember, ASP.NET generates the `_doPostBack()` function automatically, provided at least one control on the page uses automatic postbacks.

Finally, any control that has its `AutoPostBack` property set to true is connected to the `_doPostBack()` function by using the `onclick` or `onchange` attributes. These attributes indicate what action the browser should take in response to the client-side JavaScript events `onclick` and `onchange`.

The following example shows the tag for a list control named `IstBackColor`, which posts back automatically. Whenever the user changes the selection in the list, the client-side `onchange` event fires. The browser then calls the `_doPostBack()` function, which sends the page back to the server.

```
<select ID="IstBackColor" onchange="__doPostBack('IstBackColor','')"
language="javascript">
```

In other words, ASP.NET automatically changes a client-side JavaScript event into a server-side ASP.NET event, using the `_doPostBack()` function as an intermediary. Figure 6-13 shows this process.

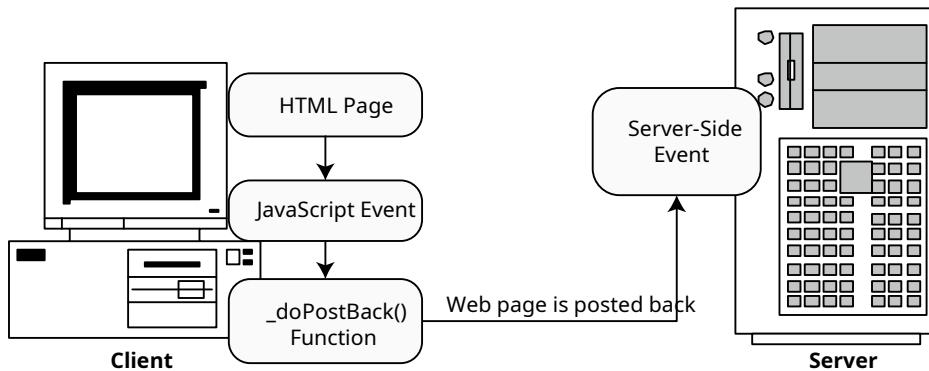


Figure 6-13. An automatic postback

The Page Life Cycle

To understand how web control events work, you need to have a solid understanding of the page life cycle. Consider what happens when a user changes a control that has the AutoPostBack property set to true:

1. On the client side, the JavaScript `_doPostBack` function is invoked, and the page is resubmitted to the server.
2. ASP.NET re-creates the Page object by using the .aspx file.
3. ASP.NET retrieves state information from the hidden view state field and updates the controls accordingly.
4. The `Page.Load` event is fired.
5. The appropriate change event is fired for the control. (If more than one control has been changed, the order of change events is undetermined.)
6. The `Page.PreRender` event fires, and the page is rendered (transformed from a set of objects to an HTML page).
7. Finally, the `Page.Unload` event is fired.
8. The new page is sent to the client.

To watch these events in action, it helps to create a simple event tracker application. All this application does is write a new entry to a list control every time one of the events it's monitoring occurs. This allows you to see the order in which events are triggered. Figure 6-14 shows what the window looks like after it's been loaded once, posted back (when the text box content was changed), and posted back again (when the check box state was changed).

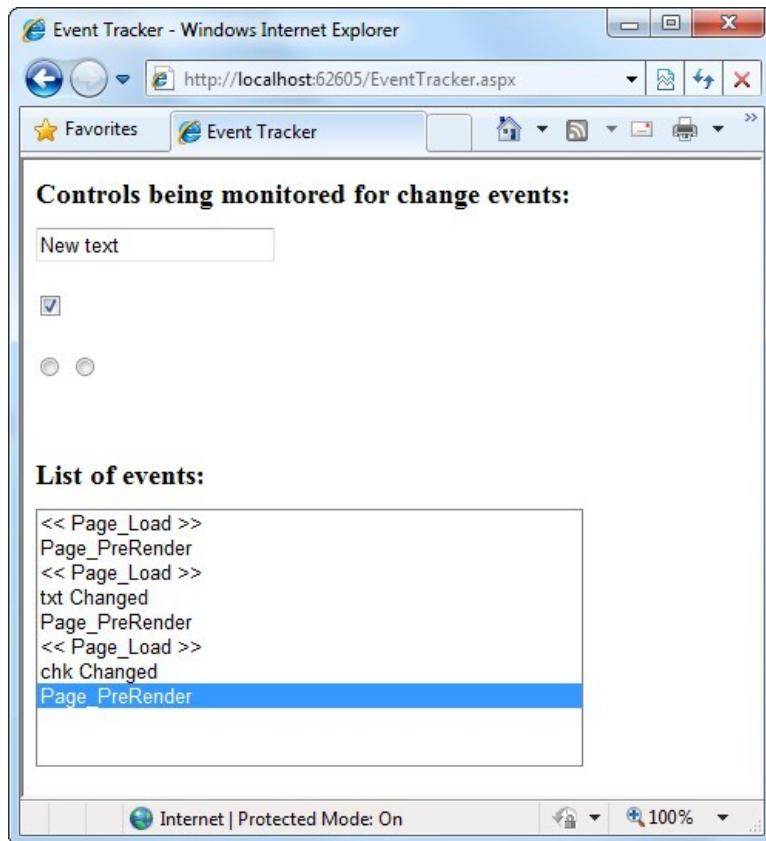


Figure 6-14. The event tracker

Listing 6-1 shows the markup code for the event tracker, and Listing 6-2 shows the code-behind class that makes it work.

Listing 6-1. EventTracker.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
   CodeFile="EventTracker.aspx.cs" Inherits="EventTracker" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <title>Event Tracker</title>
</head>
<body>
<form runat="server">
<div>
  <h1>Controls being monitored for change events:</h1>
  <asp:TextBox ID="txt" runat="server" AutoPostBack="true"
    OnTextChanged="CtrlChanged" />
  <br /><br />
```

```

<asp:CheckBox ID="chk" runat="server" AutoPostBack="true"
    OnCheckedChanged="CtrlChanged"/>
<br /><br />
<asp:RadioButton ID="opt1" runat="server" GroupName="Sample"
    AutoPostBack="True" OnCheckedChanged="CtrlChanged"/>
<asp:RadioButton ID="opt2" runat="server" GroupName="Sample"
    AutoPostBack="True" OnCheckedChanged="CtrlChanged"/>

<h1>List of events:</h1>
<asp:ListBox ID="lstEvents" runat="server" Width="355px"
    Height="150px" /><br />
<br /><br /><br />
</div>
</form>
</body>
</html>

```

Listing 6-2. EventTracker.aspx.cs

```

public partial class EventTracker : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Log("<< Page_Load >>");
    }

    protected void Page_PreRender(object sender, EventArgs e)
    {
        // When the Page.PreRender event occurs, it is too late
        // to change the list.
        Log("Page_PreRender");
    }

    protected void CtrlChanged(Object sender, EventArgs e)
    {
        // Find the control ID of the sender.
        // This requires converting the Object type into a Control class.
        string ctrlName = ((Control)sender).ID;
        Log(ctrlName + " Changed");
    }

    private void Log(string entry)
    {
        lstEvents.Items.Add(entry);

        // Select the last item to scroll the list so the most recent
        // entries are visible.
        lstEvents.SelectedIndex = lstEvents.Items.Count - 1;
    }
}

```

Dissecting the Code . . .

The following points are worth noting about this code:

- The code writes to the ListBox by using a private Log() method. The Log() method adds the text and automatically scrolls to the bottom of the list each time a new entry is added, thereby ensuring that the most recent entries remain visible.
- All the change events are handled by the same method, CtrlChanged(). If you look carefully at the .aspx file, you'll notice that each input control connects its monitored event to the CtrlChanged() method. The event-handling code in the CtrlChanged() method uses the source parameter to find out what control sent the event, and it incorporates that information in the log string.
- The page includes event handlers for the Page.Load and Page.PreRender events. As with all page events, these event handlers are connected by method name. That means to add the event handler for the Page.PreRender event, you simply need to add a method named Page_PreRender(), like the one shown here.

An Interactive Web Page

Now that you've had a whirlwind tour of the basic web control model, it's time to put it to work with the second single-page utility. In this case, it's a simple example for a dynamic e-card generator. You could extend this sample (for example, allowing users to store e-cards to the database), but even on its own, this example demonstrates basic control manipulation with ASP.NET.

The web page is divided into two regions. On the left is an ordinary <div> tag containing a set of web controls for specifying card options. On the right is a Panel control (named pnlCard), which contains two other controls (lblGreeting and imgDefault) that are used to display user-configurable text and a picture. This text and picture represents the greeting card. When the page first loads, the card hasn't yet been generated, and the right portion is blank (as shown in Figure 6-15).

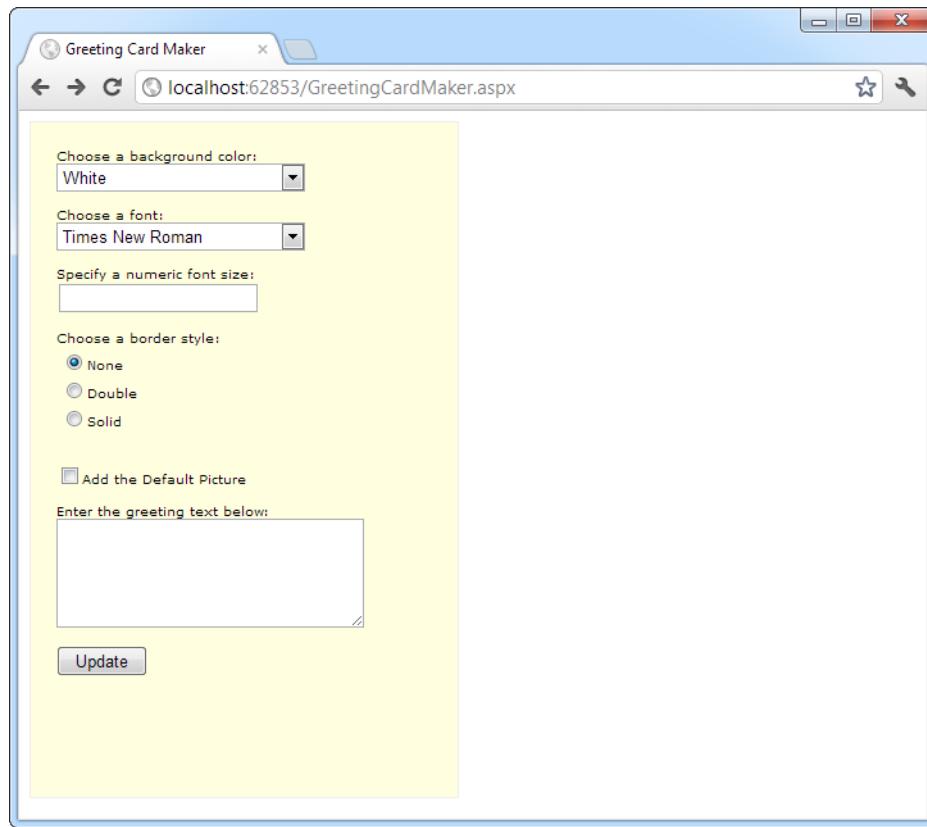


Figure 6-15. The e-card generator

Tip The `<div>` element is useful when you want to group text and controls and apply a set of formatting properties (such as a color or font) to all of them. The `<div>` element is also an essential tool for positioning content in a page. For these reasons, the `<div>` element is used in many of the examples in this book. more about using `<div>` for layout and formatting in Chapter 12.

Whenever the user clicks the Update button, the page is posted back and the “card” is updated (see Figure 6-16).

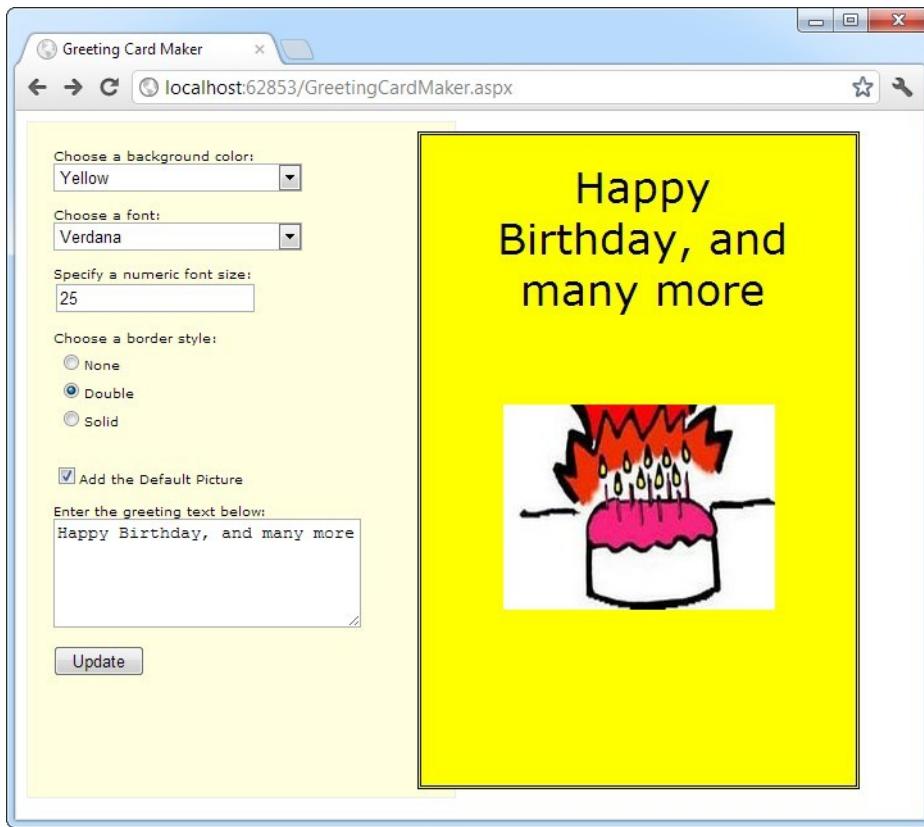


Figure 6-16. A user-configured greeting card

The .aspx layout code is straightforward. Of course, the sheer length of it makes it difficult to work with efficiently. Here's the markup, with the formatting details trimmed down to the bare essentials:

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeFile="GreetingCardMaker.aspx.cs" Inherits="GreetingCardMaker" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <title>Greeting Card Maker</title>
</head>
<body>
<form runat="server">
<div>
  <!-- Here are the controls: -->
  Choose a background color:<br />
  <asp:DropDownList ID="lstBackColor" runat="server" Width="194px"
    Height="22px"/><br /><br />
  Choose a font:<br />
  <asp:DropDownList ID="lstFontName" runat="server" Width="194px"
    Height="22px" /><br /><br />
```

```

Specify a numeric font size:<br />
<asp:TextBox ID="txtFontSize" runat="server" /><br /><br />
Choose a border style:<br />
<asp:RadioButtonList ID="lstBorder" runat="server" Width="177px"
    Height="59px" /><br /><br />
<asp:CheckBox ID="chkPicture" runat="server"
    Text="Add the Default Picture"></asp:CheckBox><br /><br />
Enter the greeting text below:<br />
<asp:TextBox ID="txtGreeting" runat="server" Width="240px" Height="85px"
    TextMode="MultiLine" /><br /><br />
<asp:Button ID="cmdUpdate" OnClick="cmdUpdate_Click"
    runat="server" Width="71px" Height="24px" Text="Update" />
</div>

<!-- Here is the card: -->
<asp:Panel ID=" pnlCard" runat="server"
    Width="339px" Height="481px" HorizontalAlign="Center"
    style="POSITION: absolute; TOP: 16px; LEFT: 313px;">
<br />&nbsp;
<asp:Label ID="lblGreeting" runat="server" Width="256px"
    Height="150px" /><br /><br /><br />
<asp:Image ID="imgDefault" runat="server" Width="212px"
    Height="160px" />
</asp:Panel>
</form>
</body>
</html>

```

To get the two-column layout in this example, you have two choices. You can use HTML tables (which are a somewhat old-fashioned technique), or you can use absolute positioning with CSS styles (as in this example). The essence of absolute positioning is easy to grasp. Just look at the style attribute in the Panel control, which specifies a fixed top and left coordinate on the web page. When the panel is rendered to HTML, this point becomes its top-left corner.

Note Absolute positioning is a feature of CSS, the Cascading style sheets standard. As such, it works in XHTML element, not just ASP.NET controls. Absolute positioning is described in detail in Chapter 12.

The code follows the familiar pattern with an emphasis on two events: the Page.Load event, where initial values are set, and the Button.Click event, where the card is generated.

using System.Drawing;

```

public partial class GreetingCardMaker : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            // Set color options.
            lstBackColor.Items.Add("White");

```

```
lstBackColor.Items.Add("Red");
lstBackColor.Items.Add("Green");
lstBackColor.Items.Add("Blue");
lstBackColor.Items.Add("Yellow");

// Set font options.
lstFontName.Items.Add("Times New Roman");
lstFontName.Items.Add("Arial");
lstFontName.Items.Add("Verdana");
lstFontName.Items.Add("Tahoma");

// Set border style options by adding a series of
// ListItem objects.
ListItem item = new ListItem();

// The item text indicates the name of the option.
item.Text = BorderStyle.None.ToString();

// The item value records the corresponding integer
// from the enumeration. To obtain this value, you
// must cast the enumeration value to an integer,
// and then convert the number to a string so it
// can be placed in the HTML page.
item.Value = ((int)BorderStyle.None).ToString();

// Add the item.
lstBorder.Items.Add(item);

// Now repeat the process for two other border styles.
item = new ListItem();
item.Text = BorderStyle.Double.ToString();
item.Value = ((int)BorderStyle.Double).ToString();
lstBorder.Items.Add(item);

item = new ListItem();
item.Text = BorderStyle.Solid.ToString();
item.Value = ((int)BorderStyle.Solid).ToString();
lstBorder.Items.Add(item);

// Select the first border option.
lstBorder.SelectedIndex = 0;

// Set the picture.
imgDefault.ImageUrl = "defaultpic.png";
}

}

protected void cmdUpdate_Click(object sender, EventArgs e)
{
    // Update the color.
    pnlCard.BackColor = Color.FromName(lstBackColor.SelectedItem.Text);
```

```

// Update the font.
lblGreeting.Font.Name = lstFontName.SelectedItem.Text;

if (Int32.Parse(txtFontSize.Text) > 0)
{
    lblGreeting.Font.Size =
        FontUnit.Point(Int32.Parse(txtFontSize.Text));
}

// Update the border style. This requires two conversion steps.
// First, the value of the list item is converted from a string
// into an integer. Next, the integer is converted to a value in
// the BorderStyle enumeration.
int borderValue = Int32.Parse(lstBorder.SelectedItem.Value);
pnlCard.BorderStyle = (BorderStyle)borderValue;

// Update the picture.
if (chkPicture.Checked)
{
    imgDefault.Visible = true;
}
else
{
    imgDefault.Visible = false;
}

// Set the text.
lblGreeting.Text = txtGreeting.Text;
}
}

```

As you can see, this example limits the user to a few preset font and color choices. The code for the `BorderStyle` option is particularly interesting. The `IstBorder` control has a list that displays the text name of one of the `BorderStyle` enumerated values. You'll remember from the introductory chapters that every enumerated value is really an integer with a name assigned to it. The `IstBorder` also secretly stores the corresponding number so that the code can retrieve the number and set the enumeration easily when the user makes a selection and the `cmdUpdate_Click` event handler fires.

Improving the Greeting Card Generator

ASP.NET pages have access to the full .NET class library. With a little exploration, you'll find classes that might help the greeting-card maker, such as tools that let you retrieve all the known color names and all the fonts installed on the web server.

For example, you can fill the `IstFontName` control with a list of fonts by using the `InstalledFontCollection` class. To access it, you need to import the `System.Drawing.Text` namespace:

```
using System.Drawing.Text;
```

Here's the code that gets the list of fonts and uses it to fill the list:

```
InstalledFontCollection fonts = new InstalledFontCollection();
foreach (FontFamily family in fonts.Families)
```

```
{  
    lstFontName.Items.Add(family.Name);  
}
```

Figure 6-17 shows the resulting font list.

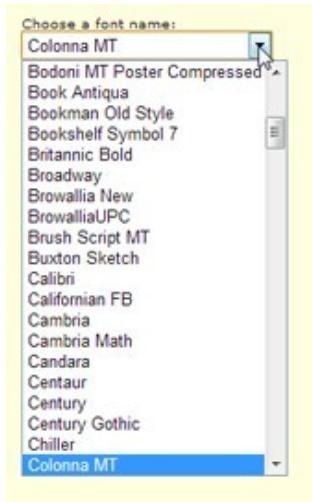


Figure 6-17. The font list

To get a list of the color names, you need to resort to a more advanced trick. Although you could hard-code a list of common colors, .NET provides a long list of color names in the System.Drawing.KnownColor enumeration. However, actually *extracting* the names from this enumeration takes some work.

The trick is to use a basic feature of all enumerations: the static Enum.GetNames() method, which inspects an enumeration and provides an array of strings, with one string for each value in the enumeration. The web page can then use data binding to automatically fill the list control with the information in the ColorArray. (You'll explore data binding in much more detail in Chapter 15.)

Note Don't worry if this example introduces a few features that look entirely alien! These features are advanced (and aren't tied specifically to ASP.NET). However, they show you some of the flavor that the framework library can provide for a mature application.

Now you can remove the code that filled the `IstBackColor` control in the previous example. Instead, you add the following code, which copies the much larger collection of color names into the list box:

```
string[] colorArray = Enum.GetNames(typeof(KnownColor));
IstBackColor.DataSource = colorArray;
IstBackColor.DataBind();
```

A minor problem with this approach is that it includes system environment colors (for example, `ActiveBorder`) in the list. It may not be obvious to the user what colors these values represent. Still, this approach works well for this simple application. You can use a similar technique to fill in `BorderStyle` options:

```
// Set border style options.
string[] borderStyleArray = Enum.GetNames(typeof(BorderStyle));
IstBorder.DataSource = borderStyleArray;
IstBorder.DataBind();
```

This code raises a new challenge: how do you convert the value that the user selects into the appropriate constant for the enumeration? When the user chooses a border style from the list, the `SelectedItem` property will have a text string such as "Groove". But to apply this border style to the control, you need a way to determine the enumerated constant that matches this text.

You can handle this problem in a few ways. (Earlier, you saw an example in which the enumeration integer was stored as a value in the list control.) In this case, the most direct approach uses an advanced feature called a `TypeConverter`. A `TypeConverter` is a special class that is able to convert from a specialized type (in this case, the `BorderStyle` enumeration) to a simpler type (such as a string), and vice versa.

To access this class, you need to import the `System.ComponentModel` namespace:

```
using System.ComponentModel;
```

You can then add the following code to the `cmdUpdate_Click` event handler:

```
// Find the appropriate TypeConverter for the BorderStyle enumeration.
TypeConverter converter =
    TypeDescriptor.GetConverter(typeof(BorderStyle));

// Update the border style using the value from the converter.
pnlCard.BorderStyle = (BorderStyle)converter.ConvertFromString(
    IstBorder.SelectedItem.Text);
```

This code gets the appropriate `TypeConverter` (in this case, one that's designed expressly to work with the `BorderStyle` enumeration). It then converts the text name (such as `Solid`) to the appropriate value (`BorderStyle.Solid`).

Generating the Cards Automatically

The last step is to use ASP.NET's automatic postback events to make the card update dynamically every time an option is changed. The `Update` button could now be used to submit the final, perfected greeting card, which might then be e-mailed to a recipient or stored in a database.

To configure the controls so they automatically trigger a page postback, simply set the `AutoPostBack` property of each input control to true. An example is shown here:

```
Choose a background color:<br />
<asp:DropDownList ID="IstBackColor" AutoPostBack="True" runat="server"
    Width="194px" Height="22px"/>
```

Next, alter the control tags so that the changed event of each input control is connected to an event handler named ControlChanged. Here's an example with the SelectedIndexChanged event or the drop-down list:

Choose a background color:


```
<asp:DropDownList ID="lstBackColor" AutoPostBack="True" runat="server"
    OnSelectedIndexChanged="ControlChanged" Width="194px" Height="22px"/>
```

You'll notice that the name of the change event depends on the control. For example, the TextBox provides a TextChanged event, the ListBox provides a SelectedIndexChanged event, and so on.

Finally, you need to create an event handler that can handle the change events. To save a few steps, you can use the same event handler for all the input controls. All the event handler needs to do is call the update routine that regenerates the greeting card.

```
protected void ControlChanged(object sender, System.EventArgs e)
{
    // Refresh the greeting card (because a control was changed).
    UpdateCard();
}

protected void cmdUpdate_Click(object sender, EventArgs e)
{
    // Refresh the greeting card (because the button was clicked).
    UpdateCard();
}

private void UpdateCard()
{
    // (The code that draws the greeting card goes here.)
}
```

With these changes, it's easy to perfect the more extensive card-generating program shown in Figure 6-18. The full code for this application is provided with the online samples.

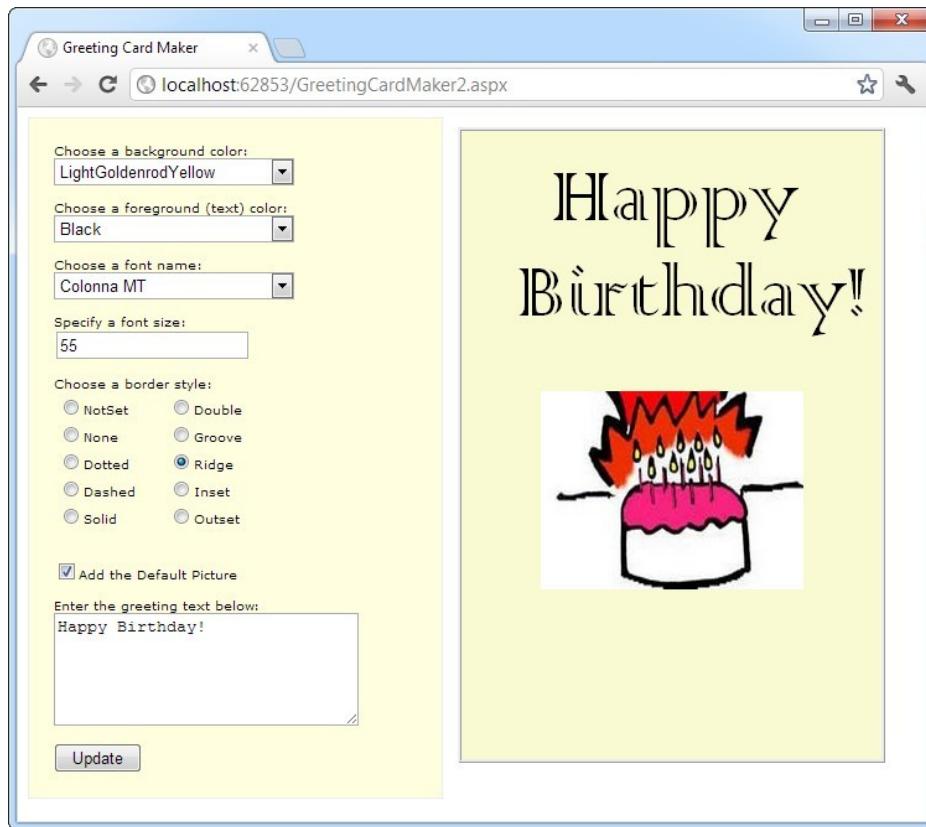


Figure 6-18. A more extensive card generator

Tip Automatic postback isn't always best. sometimes an automatic postback can annoy a user, especially if the user is working over a slow connection or when the server needs to perform a time-consuming operation. For this reason, it's sometimes best to use an explicit submit button and not enable AutoPostback for most inputs. Alternatively, you might jazz up your web page with the AsP.nET AJAX features described in Chapter 25, which allow you to create user interfaces that feel more responsive, and can update themselves without a distracting full-page refresh.

The Last Word

This chapter introduced you to one of ASP.NET's richest features: web controls, and their object interface. As you continue through this book, you'll learn about more web controls. The following highlights are still to come:

- In Chapter 10, you'll learn about advanced controls such as the AdRotator, the Calendar, and the validation controls. You'll also learn about specialized container controls, such as the MultiView and Wizard.
- In Chapter 13, you'll learn about navigation controls such as the TreeView and Menu.
- In Chapter 16, you'll learn about the GridView, DetailsView, and FormView—high-level web controls that let you manipulate a complex table of data from any data source.

For a good reference that shows each web control and lists its important properties, refer to the MSDN reference website at <http://msdn.microsoft.com/library/bb386416.aspx>.



Validation

This chapter presents some of the most useful controls that are included in ASP.NET: the *validation controls*. These controls take a potentially time-consuming and complicated task—verifying user input and reporting errors—and automate it. Each validation control, or *validator*, has its own built-in logic. Some check for missing data, others verify that numbers fall in a predefined range, and so on. In many cases, the validation controls allow you to verify user input without writing a line of code.

In this chapter, you'll learn how to use the validation controls in an ASP.NET web page and how to get the most out of them with sophisticated regular expressions, custom validation functions, and more. And as usual, you'll peer under the hood to see how ASP.NET implements these features.

Understanding Validation

As any seasoned developer knows, the people using your website will occasionally make mistakes. What's particularly daunting is the range of possible mistakes that users can make. Here are some common examples:

- A user might ignore an important field and leave it blank.
- If you disallow blank values, a user might type in semi-random nonsense to circumvent your checks. This can create endless headaches on your end. For example, you might get stuck with an invalid e-mail address that causes problems for your automatic e-mailing program.
- A user might make an honest mistake, such as entering a typing error, entering a nonnumeric character in a number field, or submitting the wrong type of information. A user might even enter several pieces of information that are individually correct but when taken together are inconsistent (for example, entering a MasterCard number after choosing Visa as the payment type).
- A malicious user might try to exploit a weakness in your code by entering carefully structured wrong values. For example, an attacker might attempt to cause a specific error that will reveal sensitive information. A more dramatic example of this technique is the *SQL injection attack*, whereby user-supplied values change the operation of a dynamically constructed database command. (Of course, validation is no defense for poor coding. When you consider database programming in Chapter 14, you'll learn how to use parameterized commands, which avoid the danger of SQL injection attacks altogether.)

A web application is particularly susceptible to these problems, because it relies on basic HTML input controls that don't have all the features of their Windows counterparts. For example, a common technique in a Windows application is to handle the KeyPress event of a text box, check to see whether the current character is valid, and prevent it from appearing if it isn't. This technique makes it easy to create a text box that accepts only numeric input.

This strategy isn't as easy in a server-side web page. To perform validation on the web server, you need to post back the page, and it just isn't practical to post the page back to the server every time the user types a letter. To avoid this sort of problem, you need to perform all your validation at once when a page (which may contain multiple input controls) is submitted. You then need to create the appropriate user interface to report the mistakes. Some websites report only the first incorrect field, while others use a table, list, or window to describe them all. By the time you've perfected your validation strategy, you'll have spent a considerable amount of effort writing tedious code.

ASP.NET aims to save you this trouble and provide you with a reusable framework of validation controls that manages validation details by checking fields and reporting on errors automatically. These controls can even use client-side JavaScript to provide a more dynamic and responsive interface while still providing ordinary validation for older browsers (often referred to as *down-level* browsers).

The Validation Controls

ASP.NET provides five validator controls, which are described in Table 9-1. Four are targeted at specific types of validation, while the fifth allows you to apply custom validation routines. You'll also see a ValidationSummary control in the Toolbox, which gives you another option for showing a list of validation error messages in one place. You'll learn about the ValidationSummary later in this chapter (see the "Other Display Options" section).

Table 9-1. Validator Controls

Control Class	Description
RequiredFieldValidator	Validation succeeds as long as the input control doesn't contain an empty string.
RangeValidator	Validation succeeds if the input control contains a value within a specific numeric, alphabetic, or date range.
CompareValidator	Validation succeeds if the input control contains a value that matches the value in another input control, or a fixed value that you specify.
RegularExpressionValidator	Validation succeeds if the value in an input control matches a specified regular expression.
CustomValidator	Validation is performed by a user-defined function.

Each validation control can be bound to a single input control. In addition, you can apply more than one validation control to the same input control to provide multiple types of validation.

If you use the RangeValidator, CompareValidator, or RegularExpressionValidator, validation will automatically succeed if the input control is empty, because there is no value to validate. If this isn't the behavior you want, you should also add a RequiredFieldValidator and link it to the same input control. This ensures that two types of validation will be performed, effectively restricting blank values.

Server-Side Validation

You can use the validator controls to verify a page automatically when the user submits it or manually in your code. The first approach is the most common.

When using automatic validation, the user receives a normal page and begins to fill in the input controls. When finished, the user clicks a button to submit the page. Every button has a CausesValidation property,

which can be set to true or false. What happens when the user clicks the button depends on the value of the CausesValidation property:

- If CausesValidation is false, ASP.NET will ignore the validation controls, the page will be posted back, and your event-handling code will run normally.
- If CausesValidation is true (the default), ASP.NET will automatically validate the page when the user clicks the button. It does this by performing the validation for each control on the page. If any control fails to validate, ASP.NET will return the page with some error information, depending on your settings. Your click event-handling code may or may not be executed—meaning you'll have to specifically check in the event handler whether the page is valid.

Based on this description, you'll realize that validation happens automatically when certain buttons are clicked. It doesn't happen when the page is posted back because of a change event (such as choosing a new value in an AutoPostBack list) or if the user clicks a button that has CausesValidation set to false. However, you can still validate one or more controls manually and then make a decision in your code based on the results. You'll learn about this process in more detail a little later (see the "Manual Validation" section).

Note Many other button-like controls that can be used to submit the page also provide the CausesValidation property. Examples include the linkButton, imageButton, and Bulletedlist. (Technically, the CausesValidation is defined by the iButtonControl interface, which all button-like controls implement.)

Client-Side Validation

In modern browsers (including Microsoft Internet Explorer, Mozilla Firefox, Apple Safari, and Google Chrome), ASP.NET automatically adds JavaScript code for client-side validation. In this case, when the user clicks a CausesValidation button, the same error messages will appear without the page needing to be submitted and returned from the server. This increases the responsiveness of your web page.

However, even if the page validates successfully on the client side, ASP.NET still revalidates it when it's received at the server. This is because it's easy for an experienced user to circumvent client-side validation. For example, a malicious user might delete the block of JavaScript validation code and continue working with the page. By performing the validation at both ends, ASP.NET makes sure your application can be as responsive as possible while also remaining secure.

HTML5 Validation

HTML5, the most modern version of the HTML language, adds new client-side validation features that can help catch errors. The problem is that HTML5 validation is inconsistent—it works differently in different browsers, and many browsers offer only partial support. (For the complete details, you can refer to the compatibility table at <http://caniuse.com/form-validation>.)

HTML5 validation has essentially the same effect as JavaScript-based validation. When the user types in data that doesn't match the expected data type or validation rule—for example, if the user puts text in a numeric field—the browser detects the problem. It prevents the form from being submitted and displays an error message next to the offending field.

You can use HTML5 validation in an ASP.NET web form, but it probably isn't the best choice, for two reasons:

On its own, HTML5 validation isn't a complete solution. If you use HTML5 validation, you'll need to add a JavaScript fallback for browsers that don't support HTML5 validation, and you'll need to perform similar checks on the server to catch data that's been tampered with. But if you use ASP.NET's validators, you'll get compatibility with all browsers and server-side checking for free.

HTML5 validation works on <input> elements, not web controls. And because web controls are designed to work with all browsers, not just those with the latest HTML5 features, they don't include HTML5-specific properties. That means there's no clean and practical way to set the HTML5 *required* attribute (for required fields) and the *pattern* attribute (for regular expression validation).

ASP.NET does support one HTML5 validation feature: the enhanced *type* attribute, which allows you to create text boxes that are intended for specific types of data (such as e-mail addresses or numeric values). You can set the type attribute through the `TextBox.TextMode` property. Here's an example:

```
<asp:TextBox id="txtAge" runat="server" TextMode="Number" />
```

However, even this feature doesn't integrate well with ASP.NET. If you're creating a form that doesn't *need* validation, and doesn't use any ASP.NET validators, this approach is perfectly reasonable. And in some browsers, it will give the user additional editing conveniences, such as a numeric-only keypad on a tablet computer (for numeric data types), or a calendar-style date-picker control (for date data types). However, if you combine this approach with ASP.NET validators, you can end up with a mishmash of error messages. For example, see Figure 9-1, where ASP.NET validation gives the message *The email is missing the @ symbol*, and Google Chrome complains, *Please enter an email address*.

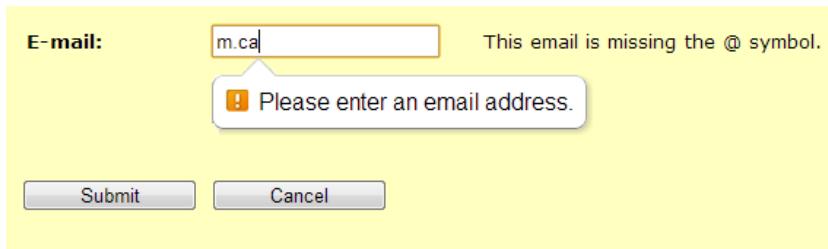


Figure 9-1. HTML5 validation and ASP.NET validation—an awkward match

Here's the bottom line: For the vast majority of ASP.NET web forms, the best solution is to use ASP.NET validators. Sometimes a tried-and-true solution is better than living on the bleeding edge.

Using the Validation Controls

The validation controls are found in the `System.Web.UI.WebControls` namespace and inherit from the `BaseValidator` class. This class defines the basic functionality for a validation control. Table 9-2 describes its key properties.

Table 9-2. Properties of the *BaseValidator* Class

Property	Description
ControlToValidate	Identifies the control that this validator will check. Each validator can verify the value in one input control. However, it's perfectly reasonable to "stack" validators—in other words, attach several validators to one input control to perform more than one type of error checking.
ErrorMessage and ForeColor	If validation fails, the validator control can display a text message (set by the ErrorMessage property). By changing the ForeColor, you can make this message stand out in angry red lettering.
Display	Allows you to configure whether this error message will be inserted into the page dynamically when it's needed (Dynamic) or whether an appropriate space will be reserved for the message (Static). Dynamic is useful when you're placing several validators next to each other. That way, the space will expand to fit the currently active error indicators, and you won't be left with any unseemly whitespace. Static is useful when the validator is in a table and you don't want the width of the cell to collapse when no message is displayed. Finally, you can also choose None to hide the error message altogether.
IsValid	After validation is performed, this returns true or false depending on whether it succeeded or failed. Generally, you'll check the state of the entire page by looking at its IsValid property instead to find out if all the validation controls succeeded.
Enabled	When set to false, automatic validation will not be performed for this control when the page is submitted.
EnableClientScript	If set to true, ASP.NET will add JavaScript and DHTML code to allow client-side validation on browsers that support it.

When using a validation control, the only properties you need to implement are ControlToValidate and ErrorMessage. In addition, you may need to implement the properties that are used for your specific validator. Table 9-3 outlines these properties.

Table 9-3. Validator-Specific Properties

Validator Control	Added Members
RequiredFieldValidator	None required
RangeValidator	MaximumValue, MinimumValue, Type
CompareValidator	ControlToCompare, Operator, Type, ValueToCompare
RegularExpressionValidator	ValidationExpression
CustomValidator	ClientValidationFunction, ValidateEmptyText, ServerValidate event

Later in this chapter (in the “A Validated Customer Form” section), you’ll see a customer form example that demonstrates each type of validation.

A Simple Validation Example

To understand how validation works, you can create a simple web page. This test uses a single Button web control, two TextBox controls, and a RangeValidator control that validates the first text box. If validation fails, the RangeValidator control displays an error message, so you should place this control immediately next to the TextBox it’s validating. The second text box does not use any validation.

Figure 9-2 shows the appearance of the page after a failed validation attempt.

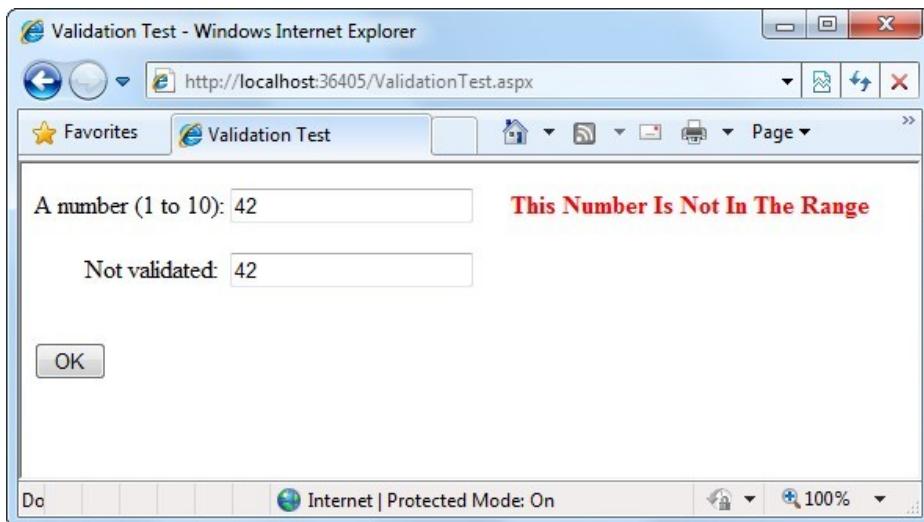


Figure 9-2. Failed validation

In addition, place a Label control at the bottom of the form. This label will report when the page has been posted back and the event-handling code has executed. Disable its EnableViewState property to ensure that it will be cleared every time the page is posted back.

The markup for this page defines a RangeValidator control, sets the error message, identifies the control that will be validated, and requires an integer from 1 to 10. These properties are set in the .aspx file, but they could also be configured in the event handler for the Page.Load event. The Button automatically has its CauseValidation property set to true, because this is the default.

A number (1 to 10):

```
<asp:TextBox id="txtValidated" runat="server" />
<asp:RangeValidator id="RangeValidator" runat="server"
    ErrorMessage="This Number Is Not In The Range"
    ControlToValidate="txtValidated"
    MaximumValue="10" MinimumValue="1"
    ForeColor="Red" Font-Bold="true"
    Type="Integer" />
<br /><br />
```

Not validated:

```
<asp:TextBox id="txtNotValidated" runat="server" /><br /><br />
<asp:Button id="cmdOK" runat="server" Text="OK" OnClick="cmdOK_Click" />
<br /><br />
<asp:Label id="lblMessage" runat="server"
    EnableViewState="False" />
```

Finally, here is the code that responds to the button click:

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
    lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

If you're testing this web page in a modern browser, you'll notice an interesting trick. When you first open the page, the error message is hidden. But if you type an invalid number (remember, validation will succeed for an empty value) and press the Tab key to move to the second text box, an error message will appear automatically next to the offending control. This is because ASP.NET adds a special JavaScript function that detects when the focus changes. The actual implementation of this JavaScript code is somewhat complicated, but ASP.NET handles all the details for you automatically. As a result, if you try to click the OK button with an invalid value in txtValidated, your actions will be ignored, and the page won't be posted back.

Not all browsers will support client-side validation. To see what will happen on a down-level browser, set the RangeValidator.EnableClientScript property to false, and rerun the page. Now error messages won't appear dynamically as you change focus. However, when you click the OK button, the page will be returned from the server with the appropriate error message displayed next to the invalid control.

The potential problem in this scenario is that the click event-handling code will still execute, even though the page is invalid. To correct this problem and ensure that your page behaves the same on modern and older browsers, you must specifically abort the event code if validation hasn't been performed successfully.

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
    // Abort the event if the control isn't valid.
    if (!RangeValidator.IsValid) return;

    lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

This code solves the current problem, but it isn't much help if the page contains multiple validation controls. Fortunately, every web form provides its own IsValid property. This property will be false if *any* validation control has failed. It will be true if all the validation controls completed successfully. If validation was not performed (for example, if the validation controls are disabled or if the button has CausesValidation set to false), you'll get an HttpException when you attempt to read the IsValid property.

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
    // Abort the event if any control on the page is invalid.
    if (!Page.IsValid) return;

    lblMessage.Text = "cmdOK_Click event handler executed.";
}
```

Remember, client-side validation is just nice frosting on top of your application. Server-side validation will always be performed, ensuring that crafty users can't "spoof" pages.

Other Display Options

In some cases, you might have already created a carefully designed form that combines multiple input fields. Perhaps you want to add validation to this page, but you can't reformat the layout to accommodate all the error messages for all the validation controls. In this case, you can save some work by using the ValidationSummary control.

To try this, set the Display property of the RangeValidator control to None. This ensures that the error message will never be displayed. However, validation will still be performed and the user will still be prevented from successfully clicking the OK button if some invalid information exists on the page.

Next, add the ValidationSummary in a suitable location (such as the bottom of the page):

```
<asp:ValidationSummary id="Errors" runat="server" ForeColor="Red" />
```

When you run the page, you won't see any dynamic messages as you enter invalid information and tab to a new field. However, when you click the OK button, the ValidationSummary will appear with a list of all error messages, as shown in Figure 9-3. In this case, it retrieves one error message (from the RangeValidator control). However, if you had a dozen validators, it would retrieve all their error messages and create a list.

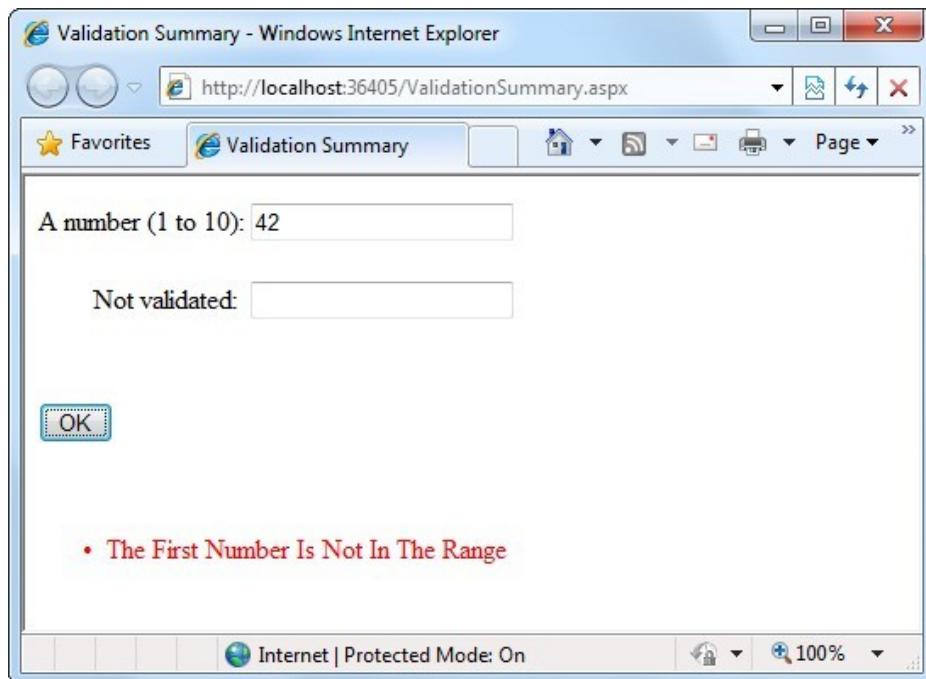


Figure 9-3. The validation summary

When the ValidationSummary displays the list of errors, it automatically retrieves the value of the ErrorMessage property from each validator. In some cases, you'll want to display a full message in the summary and some sort of visual indicator next to the offending control. For example, many websites use an error icon or an asterisk to highlight text boxes with invalid input. You can use this technique with the help of the Text property of the validators. Ordinarily, Text is left empty, and the validator doesn't show any content in the web page. However, if you set both Text and ErrorMessage, the ErrorMessage value will be used for the summary while the

Text value is displayed in the validator. (Of course, you'll need to make sure you aren't also setting the Display property of your validator to None, which hides the validator—and its content—completely.)

Here's an example of a validator that includes a detailed error message (which will appear in the ValidationSummary) and an asterisk indicator (which will appear in the validator, next to the control that has the problem):

```
<asp:RangeValidator id="RangeValidator" runat="server"
    Text="* Error Message=The First Number Is Not In The Range"
    ControlToValidate="txtValidated"
    MaximumValue="10" MinimumValue="1" Type="Integer" />
```

If you have a lot of text, you may prefer to nest it inside the `<asp:RangeValidator>` element. For example, you can rewrite the markup shown earlier with this:

```
<asp:RangeValidator id="RangeValidator" runat="server"
    ControlToValidate="txtValidated" MaximumValue="10" MinimumValue="1"
    ErrorMessage="The First Number Is Not In The Range"
    Type="Integer"><b>*** Error</b></asp:RangeValidator>
```

Here, ASP.NET automatically extracts the HTML inside the `<asp:RangeValidator>` element and uses it to set the `RangeValidator.Text` property.

You can even get a bit fancier by replacing the plain asterisk with a snippet of more-interesting HTML. Here's an example that uses the `` tag to add a small error icon image when validation fails:

```
<asp:RangeValidator id="RangeValidator" runat="server">
    
</asp:RangeValidator>
```

Figure 9-4 shows this validator in action.

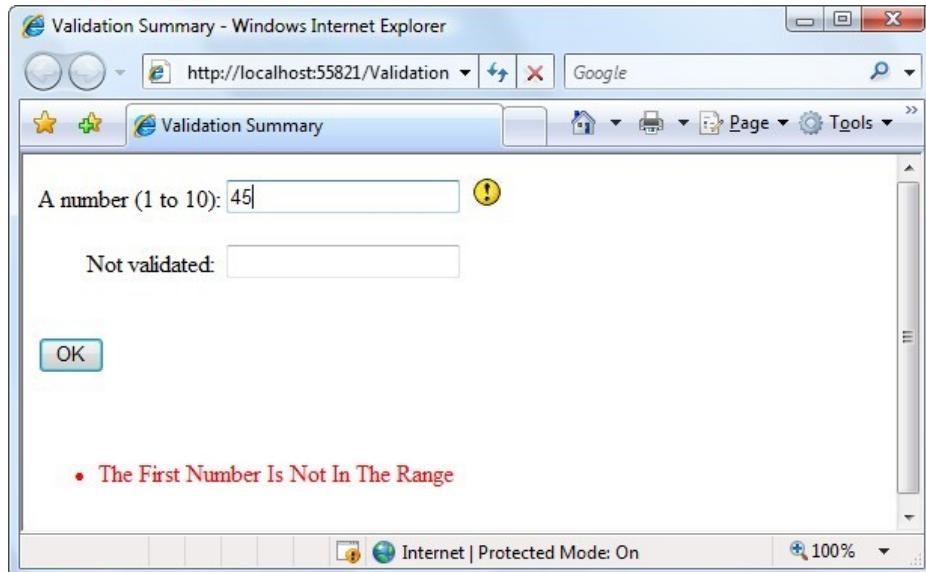


Figure 9-4. A validation summary and an error indicator

The ValidationSummary control provides some useful properties you can use to fine-tune the error display. You can set the HeaderText property to display a special title at the top of the list (such as *Your page contains the following errors:*). You can also change the ForeColor and choose a DisplayMode. The possible modes are BulletList (the default), List, and SingleParagraph.

Finally, you can choose to have the validation summary displayed in a pop-up dialog box instead of on the page (see Figure 9-4). This approach has the advantage of leaving the user interface of the page untouched, but it also forces the user to dismiss the error messages by closing the window before being able to modify the input controls. If users will need to refer to these messages while they fix the page, the inline display is better.

To show the summary in a dialog box, set the ShowMessageBox property of the ValidationSummary to true. Keep in mind that unless you set the ShowSummary property to false, you'll see both the message box and the in-page summary (as in Figure 9-5).

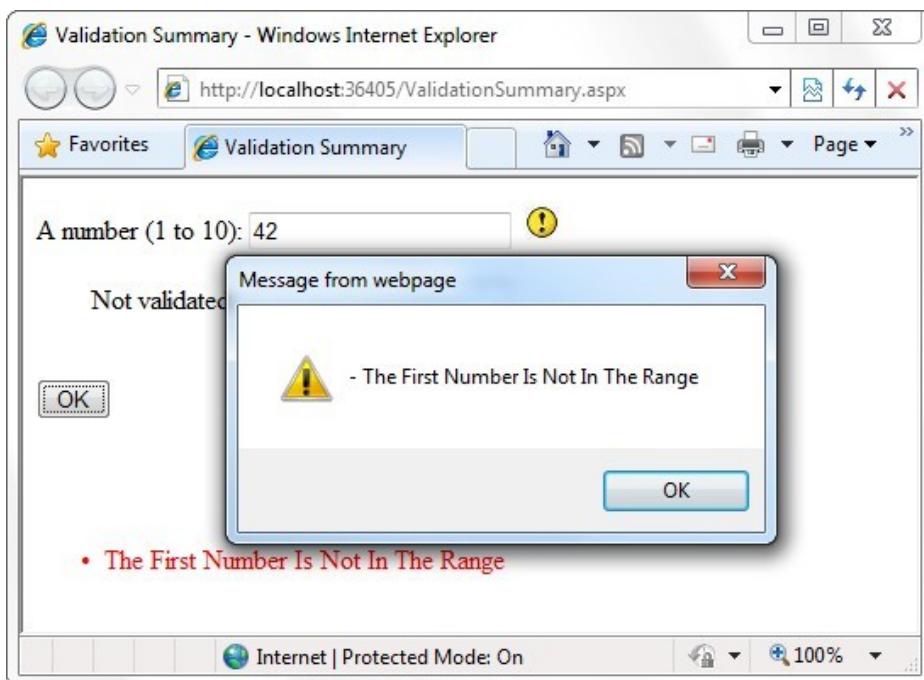


Figure 9-5. A message box summary

Manual Validation

Your final option is to disable validation and perform the work on your own, with the help of the validation controls. This allows you to take other information into consideration or create a specialized error message that involves other controls (such as images or buttons).

You can create manual validation in one of three ways:

- Use your own code to verify values. In this case, you won't use any of the ASP.NET validation controls.
- Disable the EnableClientScript property for each validation control. This allows an invalid page to be submitted, after which you can decide what to do with it depending on the problems that may exist.

- Add a button with CausesValidation set to false. When this button is clicked, manually validate the page by calling the Page.Validate() method. Then examine the IsValid property and decide what to do.

The next example uses the second approach. After the page is submitted, it examines all the validation controls on the page by looping through the PageValidators collection. Every time it finds a control that hasn't validated successfully, it retrieves the invalid value from the input control and adds it to a string. At the end of this routine, it displays a message that describes which values were incorrect, as shown in Figure 9-6.

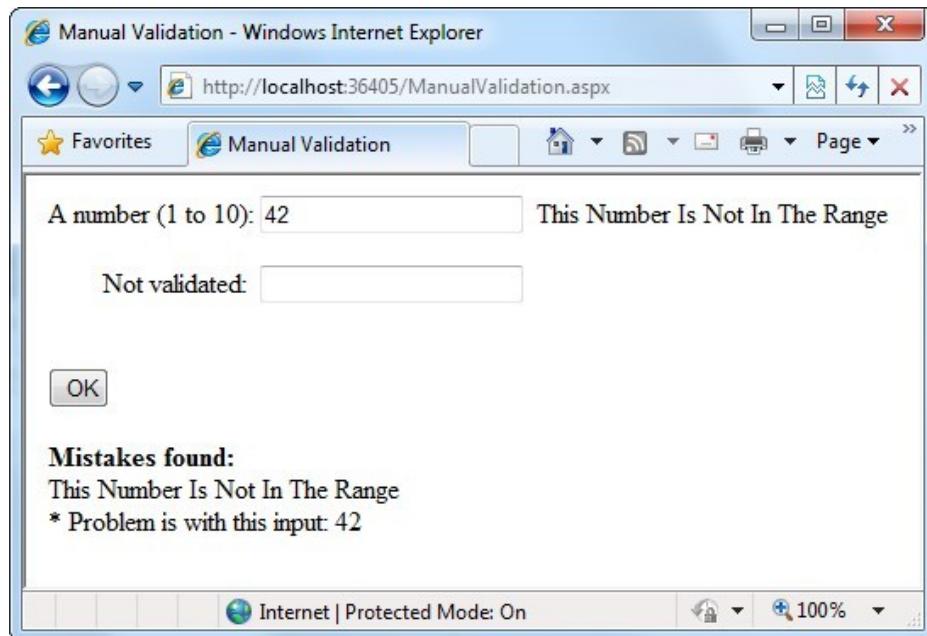


Figure 9-6. Manual validation

This technique adds a feature that wouldn't be available with automatic validation, which uses the ErrorMessage property. In that case, it isn't possible to include the actual incorrect values in the message.

To try this example, set the EnableClientScript property of each validator to false. Then you can use the code in this event handler to check for invalid values.

```
protected void cmdOK_Click(Object sender, EventArgs e)
{
    string errorMessage = "<b>Mistakes found:</b><br />";

    // Search through the validation controls.
    foreach (BaseValidator ctrl in thisValidators)
    {
        if (!ctrl.IsValid)
        {
            errorMessage += ctrl.ErrorMessage + "<br />";
        }
    }
}
```

```

// Find the corresponding input control, and change the
// generic Control variable into a TextBox variable.
// This allows access to the Text property.
TextBox ctrlInput =
    (TextBox)this.FindControl(ctrl.ControlToValidate);
errorMessage += " * Problem is with this input: ";
errorMessage += ctrlInput.Text + "<br />";
}
}
lblMessage.Text = errorMessage;
}

```

This example uses an advanced technique: the `Page.FindControl()` method. It's required because the `ControlToValidate` property of each validator simply provides a string with the name of a control, not a reference to the actual control object. To find the control that matches this name (and retrieve its `Text` property), you need to use the `FindControl()` method. After retrieving the matching text box, the code can perform other tasks such as clearing the current value, tweaking a property, or even changing the text box color. Note that the `FindControl()` method returns a generic `Control` reference, because you might search any type of control. To access all the properties of your control, you need to cast it to the appropriate type (such as `TextBox` in this example).

Tip in this example, the code finds each validator and reads the `ErrorMessage` property. However, you set the `ErrorMessage` property at this time, which allows you to create customized error messages that include information about the invalid values in their text.

Validation with Regular Expressions

One of ASP.NET's most powerful validation controls is the `RegularExpressionValidator`, which validates text by determining whether it matches a specific pattern.

For example, e-mail addresses, phone numbers, and file names are all examples of text that has specific constraints. A phone number must be a set number of digits, an e-mail address must include exactly one @ character (with text on either side), and a file name can't include certain special characters such as \ and ?. One way to define patterns like these is with *regular expressions*.

Regular expressions have appeared in countless other languages and gained popularity as an extremely powerful way to work with strings. In fact, Visual Studio even allows programmers to perform a search-and-replace operation in their code using a regular expression (which may represent a new height of computer geekdom). Regular expressions can almost be considered an entire language of their own. How to master all the ways you can use regular expressions—including pattern matching, back references, and named groups—could occupy an entire book (and several books are dedicated to just that subject). Fortunately, you can understand the basics of regular expressions without nearly that much work.

Using Literals and Metacharacters

All regular expressions consist of two kinds of characters: literals and metacharacters. *Literals* are not unlike the string literals you type in code. They represent a specific defined character. For example, if you search for the string literal "a", you'll find the character *a* and nothing else.

Metacharacters provide the true secret to unlocking the full power of regular expressions. You're probably already familiar with two metacharacters from the DOS world (?) and (*). Consider the command-line expression shown here:

Del *.*

The expression *.* contains one literal (the period) and two metacharacters (the asterisks). This translates as "delete every file that starts with any number of characters and ends with an extension of any number of characters (or has no extension at all)." Because all files in DOS implicitly have extensions, this has the well-documented effect of deleting everything in the current directory.

Another DOS metacharacter is the question mark, which means "any single character." For example, the following statement deletes any file named *hello* that has an extension of exactly one character.

Del hello.?

The regular expression language provides many flexible metacharacters—far more than the DOS command line. For example, \s represents any whitespace character (such as a space or tab). \d represents any digit. Thus, the following expression would match any string that started with the numbers 333, followed by a single whitespace character and any three numbers. Valid matches would include 333 333 and 333 945 but not 334 333 or 3334 945.

333\s\d\d\d\d

One aspect that can make regular expressions less readable is that they use special metacharacters that are more than one character long. In the previous example, \s represents a single character, as does \d, even though they both occupy two characters in the expression.

You can use the plus (+) sign to represent a repeated character. For example, 5+7 means "one or more occurrences of the character 5, followed by a single 7." The number 57 would match, as would 555557. You can also use parentheses to group a subexpression. For example, (52)+7 would match any string that started with a sequence of 52. Matches would include 527, 52527, 5252527, and so on.

You can also delimit a range of characters by using square brackets. [a-f] would match any single character from *a* to *f* (lowercase only). The following expression would match any word that starts with a letter from *a* to *f*, contains one or more "word" characters (letters), and ends with *ing*—possible matches include *acting* and *developing*.

[a-f]\w+ing

The following is a more useful regular expression that can match any e-mail address by verifying that it contains the @ symbol. The dot is a metacharacter used to indicate any character except a newline. However, some invalid e-mail addresses would still be allowed, including those that contain spaces and those that don't include a dot (.). You'll see a better example a little later in the customer form example.

.+@.+

Finding a Regular Expression

Clearly, picking the perfect regular expression may require some testing. In fact, numerous reference materials (on the Internet and in paper form) include useful regular expressions for validating common values such as postal codes. To experiment, you can use the simple RegularExpressionTest page included with the online samples, which is shown in Figure 9-7. It allows you to set a regular expression that will be used to validate a control. Then you can type in some sample values and see whether the regular expression validator succeeds or fails.

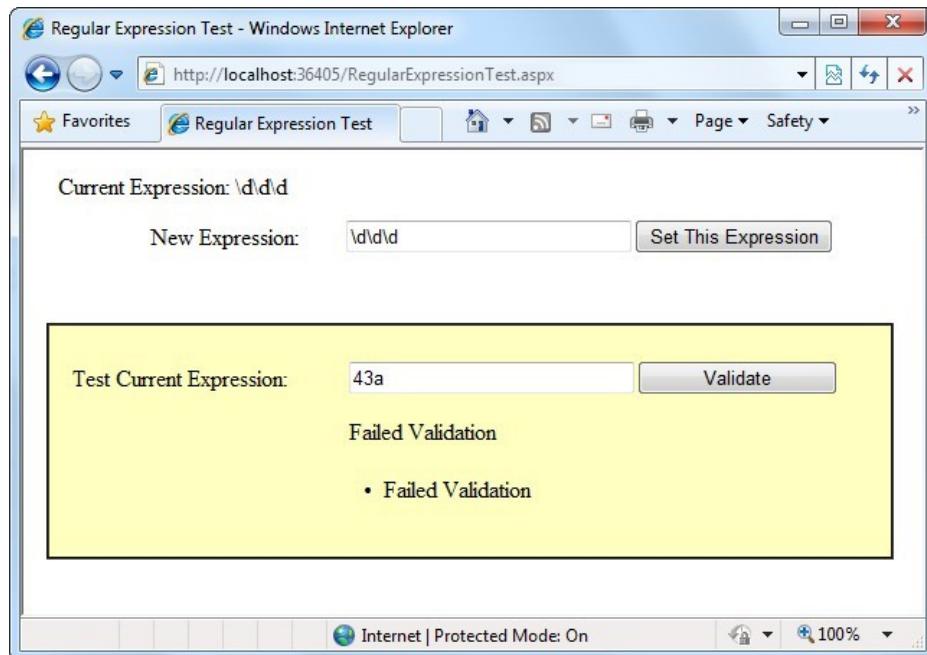


Figure 9-7. A regular expression test page

The code is quite simple. The Set This Expression button assigns a new regular expression to the RegularExpressionValidator control (using whatever text you have typed). The Validate button simply triggers a postback, which causes ASP.NET to perform validation automatically. If an error message appears, validation has failed. Otherwise, it's successful.

```

public partial class RegularExpressionTest : System.Web.UI.Page
{
    protected void cmdSetExpression_Click(Object sender, EventArgs e)
    {
        TestValidator.ValidationExpression = txtExpression.Text;
        lblExpression.Text = "Current Expression: ";
        lblExpression.Text += txtExpression.Text;
    }
}

```

Table 9-4 shows some of the fundamental regular expression building blocks. If you need to match a literal character with the same name as a special character, you generally precede it with a \ character. For example, *hello* matches *hello* in a string, because the special asterisk (*) character is preceded by a slash (\).

Table 9-4. Regular Expression Characters

Character	Description
*	Zero or more occurrences of the previous character or subexpression. For example, 7*8 matches 7778 or just 8.
+	One or more occurrences of the previous character or subexpression. For example, 7+8 matches 7778 but not 8.
()	Groups a subexpression that will be treated as a single element. For example, (78)+ matches 78 and 787878.
{m,n}	The previous character (or subexpression) can occur from <i>m</i> to <i>n</i> times. For example, A{1,3} matches A, AA, or AAA.
	Either of two matches. For example, 8 6 matches 8 or 6.
[]	Matches one character in a range of valid characters. For example, [A-C] matches A, B, or C.
[^]	Matches a character that isn't in the given range. For example, [^A-B] matches any character except A and B.
.	Any character except a newline. For example, .here matches where and there.
\s	Any whitespace character (such as a tab or space).
\S	Any nonwhitespace character.
\d	Any digit character.
\D	Any character that isn't a digit.
\w	Any "word" character (letter, number, or underscore).
\W	Any character that isn't a "word" character (letter, number, or underscore).

Table 9-5 shows a few common (and useful) regular expressions.

Table 9-5. Commonly Used Regular Expressions

Content	Regular Expression	Description
E-mail address*	\S+@\S+\.\S+	Check for an at (@) sign and dot (.) and allow nonwhitespace characters only.
Password	\w+	Any sequence of one or more word characters (letter, space, or underscore).
Specific-length password	\w{4,10}	A password that must be at least four characters long but no longer than ten characters.
Advanced password	[a-zA-Z]\w{3,9}	As with the specific-length password, this regular expression will allow four to ten total characters. The twist is that the first character must fall in the range of a-z or A-Z (that is to say, it must start with a nonaccented ordinary letter).

(continued)

Table 9-5. (continued)

Content	Regular Expression	Description
Another advanced password	[a-zA-Z]\w*\d+\w*	This password starts with a letter character, followed by zero or more word characters, one or more digits, and then zero or more word characters. In short, it forces a password to contain one or more numbers somewhere inside it. You could use a similar pattern to require two numbers or any other special character.
Limited-length field	\S{4,10}	Like the password example, this allows four to ten characters, but it allows special characters (asterisks, ampersands, and so on).
US Social Security number	\d{3}-\d{2}-\d{4}	A sequence of three, two, and then four digits, with each group separated by a dash. You could use a similar pattern when requiring a phone number.

* You have many ways to validate e-mail addresses with regular expressions of varying complexity. See www.4guysfromrolla.com/webtech/validateemail.shtml for a discussion of the subject and numerous examples.

Some logic is much more difficult to model in a regular expression. An example is the Luhn algorithm, which verifies credit card numbers by first doubling every second digit, and then adding these doubled digits together, and finally dividing the sum by ten. The number is valid (although not necessarily connected to a real account) if there is no remainder after dividing the sum. To use the Luhn algorithm, you need a CustomValidator control that runs this logic on the supplied value. (You can find a detailed description of the Luhn algorithm at http://en.wikipedia.org/wiki/Luhn_formula.)

A Validated Customer Form

To bring together these various topics, you'll now see a full-fledged web form that combines a variety of pieces of information that might be needed to add a user record (for example, an e-commerce site shopper or a content site subscriber). Figure 9-8 shows this form.

The screenshot shows a Windows Internet Explorer window titled "Customer Form - Windows Internet Explorer". The URL is "http://localhost:36405/CustomerForm.aspx". The page contains a form with six fields: "User Name", "Password", "Password (retype)", "E-mail", "Age", and "Referrer Code". The "User Name" field has an error message: "You must enter a user name.". The "Password" and "Password (retype)" fields have an error message: "Your password does not match.". The "E-mail" field has an error message: "This email is missing the @ symbol.". The "Age" field has an error message: "This age is not between 0 and 120.". The "Referrer Code" field has an error message: "Try a string that starts with 014.". Below the form are two buttons: "Submit" and "Cancel". At the bottom of the browser window, the status bar shows "Done", "Internet | Protected Mode: On", and "100%".

Figure 9-8. A sample customer form

Several types of validation are taking place on the customer form:

- Three RequiredFieldValidator controls make sure the user enters a username, a password, and a password confirmation.
- A CompareValidator ensures that the two versions of the masked password match.
- A RegularExpressionValidator checks that the e-mail address contains an at (@) symbol.
- A RangeValidator ensures the age is a number from 0 to 120.
- A CustomValidator performs a special validation on the server of a “referrer code.” This code verifies that the first three characters make up a number that is divisible by 7.

The tags for the validator controls are as follows:

```
<asp:RequiredFieldValidator id="vldUserName" runat="server"
    ErrorMessage="You must enter a user name."
    ControlToValidate="txtUserName" />

<asp:RequiredFieldValidator id="vldPassword" runat="server"
    ErrorMessage="You must enter a password."
    ControlToValidate="txtPassword" />

<asp:CompareValidator id="vldRetype" runat="server"
    ErrorMessage="Your password does not match."
    ControlToCompare="txtPassword" ControlToValidate="txtRetype" />

<asp:RequiredFieldValidator id="vldRetypeRequired" runat="server"
    ErrorMessage="You must confirm your password."
    ControlToValidate="txtRetype" />
```

```
<asp:RegularExpressionValidator id="vldEmail" runat="server"
    ErrorMessage="This email is missing the @ symbol."
    ValidationExpression=".+@.+" ControlToValidate="txtEmail" />

<asp:RangeValidator id="vldAge" runat="server"
    ErrorMessage="This age is not between 0 and 120." Type="Integer"
    MinimumValue="0" MaximumValue="120"
    ControlToValidate="txtAge" />

<asp:CustomValidator id="vldCode" runat="server"
    ErrorMessage="Try a string that starts with 014."
    ValidateEmptyText="False"
    OnServerValidate="vldCode_ServerValidate"
    ControlToValidate="txtCode" />
```

The form provides two validation buttons—one that requires validation and one that allows the user to cancel the task gracefully:

```
<asp:Button id="cmdSubmit" runat="server"
    OnClick="cmdSubmit_Click" Text="Submit"></asp:Button>
<asp:Button id="cmdCancel" runat="server"
    CausesValidation="False" OnClick="cmdCancel_Click" Text="Cancel">
</asp:Button>
```

Here's the event-handling code for the buttons:

```
protected void cmdSubmit_Click(Object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        lblMessage.Text = "This is a valid form.";
    }
}

protected void cmdCancel_Click(Object sender, EventArgs e)
{
    lblMessage.Text = "No attempt was made to validate this form.";
}
```

The only form-level code that is required for validation is the custom validation code. The validation takes place in the event handler for the CustomValidator.ServerValidate event. This method receives the value it needs to validate (e.Value) and sets the result of the validation to true or false (e.IsValid).

```
protected void vldCode_ServerValidate(Object source, ServerValidateEventArgs e)
{
    try
    {
        // Check whether the first three digits are divisible by seven.
        int val = Int32.Parse(e.Value.Substring(0, 3));
        if (val % 7 == 0)
        {
            e.IsValid = true;
        }
        else
```

```

    {
        e.IsValid = false;
    }
}
catch
{
    // An error occurred in the conversion.
    // The value is not valid.
    e.IsValid = false;
}
}

```

This example also introduces one new detail: error handling. This error-handling code ensures that potential problems are caught and dealt with appropriately. Without error handling, your code may fail, leaving the user with nothing more than a cryptic error page. The reason this example requires error-handling code is that it performs two steps that aren't guaranteed to succeed. First, the Int32.Parse() method attempts to convert the data in the text box to an integer. An error will occur during this step if the information in the text box is nonnumeric (for example, if the user entered the characters 4 G). Similarly, the String.Substring() method, which extracts the first three characters, will fail if fewer than three characters appear in the text box. To guard against these problems you can specifically check these details before you attempt to use the Parse() and Substring() methods, or you can use error handling to respond to problems after they occur. (Another option is to use the TryParse() method, which returns a Boolean value that tells you whether the conversion succeeded. You saw TryParse() at work in Chapter 5.)

Tip in some cases, you might be able to replace custom validation with a particularly ingenious use of expression. However, you can use custom validation to ensure that validation code is executed only at the That prevents users from seeing your regular expression template (in the rendered JavaScript code) and to determine how they can outwit your validation routine. A user who does not have a valid credit card number, for example, could create a false one more easily if that user knew the algorithm you use to test credit card numbers.

The CustomValidator has another quirk. You'll notice that your custom server-side validation isn't performed until the page is posted back. This means that if you enable the client script code (the default), dynamic messages will appear, informing the user when the other values are incorrect—but they will not indicate any problem with the referral code until the page is posted back to the server.

This isn't really a problem, but if it troubles you, you can use the CustomValidator.ClientValidationFunction property. Add a client-side JavaScript function to the .aspx portion of the web page. Remember, you can't use client-side ASP.NET code, because C# and VB aren't recognized by the client browser.

Your JavaScript function will accept two parameters (in true .NET style), which identify the source of the event and the additional validation parameters. In fact, the client-side event is modeled on the .NET ServerValidate event. Just as you did in the ServerValidate event handler, in the client validation function, you retrieve the value to validate from the Value property of the event argument object. You then set the IsValid property to indicate whether validation succeeds or fails.

The following is the client-side equivalent for the code in the ServerValidate event handler. The JavaScript code resembles C# superficially.

```

<script type="text/javascript">
function MyCustomValidation(objSource, objArgs)
{
    // Get value.
    var number = objArgs.Value;
}

```

```
// Check value and return result.
number = number.substr(0, 3);
if (number % 7 == 0)
{
    objArgs.IsValid = true;
}
else
{
    objArgs.IsValid = false;
}
</script>
```

You can place this block of JavaScript code in the <head> section of your page. (Or, you can put in a separate file and reference that file by using a <script> tag in the <head> section, just as you would with any other JavaScript library.)

After you've added the validation script function, you must set the ClientValidationFunction property of the CustomValidator control to the name of the function. You can edit the CustomValidator tag by hand or use the Properties window in Visual Studio.

```
<asp:CustomValidator id="vldCode" runat="server"
    ErrorMessage="Try a string that starts with 014."
    ControlToValidate="txtCode"
    OnServerValidate="vldCode_ServerValidate"
    ClientValidationFunction="MyCustomValidation" />
```

ASP.NET will now call this function on your behalf when it's required.

Tip Even when you use client-side validation, you must still include the ServerValidate event handler, provide server-side validation for clients that don't support the required JavaScript and dHTML features and prevent clients from circumventing your validation by modifying the HTML page they receive.

By default, custom validation isn't performed on empty values. However, you can change this behavior by setting the CustomValidator.ValidateEmptyText property to true. This is a useful approach if you create a more detailed JavaScript function (for example, one that updates with additional information) and want it to run when the text is cleared.

Validation Groups

In more-complex pages, you might have several distinct groups of controls, possibly in separate panels. In these situations, you may want to perform validation separately. For example, you might create a form that includes a box with login controls and a box underneath it with the controls for registering a new user. Each box includes its own submit button, and depending on which button is clicked, you want to perform the validation just for that section of the page.

This scenario is possible thanks to a feature called *validation groups*. To create a validation group, you need to put the input controls, the validators, and the CausesValidation button controls into the same logical group. You do this by setting the ValidationGroup property of every control with the same descriptive string (such as "LoginGroup" or "NewUserGroup"). Every control that provides a CausesValidation property also includes the ValidationGroup property.

For example, the following page defines two validation groups, named Group1 and Group2. The controls for each group are placed into separate Panel controls.

```
<form id="form1" runat="server">
  <asp:Panel ID="Panel1" runat="server">
    <asp:TextBox ID="TextBox1" ValidationGroup="Group1" runat="server" />
    <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
      ErrorMessage="*Required" ValidationGroup="Group1"
      runat="server" ControlToValidate="TextBox1" />
    <asp:Button ID="Button1" Text="Validate Group1"
      ValidationGroup="Group1" runat="server" />
  </asp:Panel>
  <br />
  <asp:Panel ID="Panel2" runat="server">
    <asp:TextBox ID="TextBox2" ValidationGroup="Group2"
      runat="server" />
    <asp:RequiredFieldValidator ID="RequiredFieldValidator2"
      ErrorMessage="*Required" ValidationGroup="Group2"
      ControlToValidate="TextBox2" runat="server" />
    <asp:Button ID="Button2" Text="Validate Group2"
      ValidationGroup="Group2" runat="server" />
  </asp:Panel>
</form>
```

If you click the button in the topmost panel, only the first text box is validated. If you click the button in the second panel, only the second text box is validated (as shown in Figure 9-9).

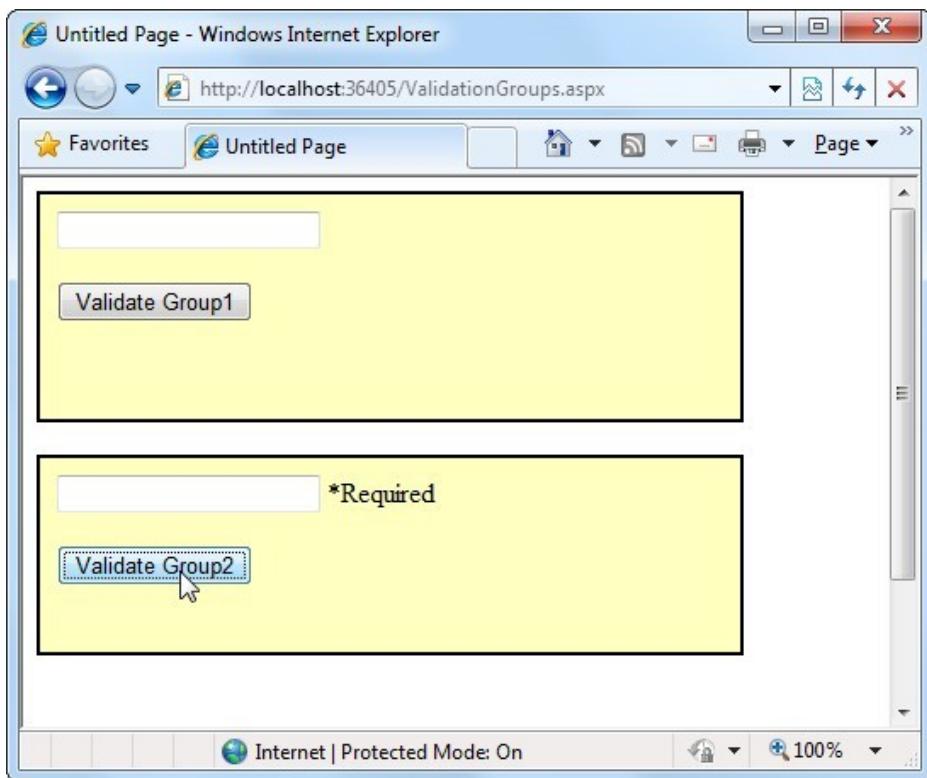


Figure 9-9. Grouping controls for validation

What happens if you add a new button that doesn't specify any validation group? In this case, the button validates every control that isn't explicitly assigned to a named validation group. In the current example, no controls fit the requirement, so the page is posted back successfully and deemed to be valid.

If you want to make sure a control is always validated, regardless of the validation group of the button that's clicked, you'll need to create multiple validators for the control, one for each group (and one with no validation group).

The Last Word

In this chapter, you learned how to use one of ASP.NET's most practical features: validation. You saw how ASP.NET combines server-side and client-side validation to ensure bulletproof security without sacrificing the usability of your web pages. You also looked at the types of validation provided by the various validation controls, and even brushed up on the powerful pattern-matching syntax used for regular expressions. Finally, you considered how to customize and extend the validation process to handle a few different scenarios.



Rich Controls

Rich controls are web controls that model complex user interface elements. Although no strict definition exists for what is and what isn't a rich control, the term commonly describes a web control that has an object model that's distinctly separate from the HTML it generates. A typical rich control can be programmed as a single object (and added to a web page with a single control tag) but renders itself using a complex sequence of HTML elements. Rich controls can also react to user actions (such as a mouse click on a specific region of the control) and raise more-meaningful events that your code can respond to on the web server. In other words, rich controls give you a way to create advanced user interfaces in your web pages without writing lines of convoluted HTML.

In this chapter, you'll take a look at several web controls that have no direct equivalent in the world of ordinary HTML. You'll start with the Calendar, which provides slick date-selection functionality. Next you'll consider the AdRotator, which gives you an easy way to insert a randomly selected image into a web page. Finally, you'll learn how to create sophisticated pages with multiple views by using two advanced controls: the MultiView and the Wizard. These controls allow you to pack a miniature application into a single page. Using them, you can handle a multistep task without redirecting the user from one page to another.

Note ASP.NET includes numerous rich controls that are discussed elsewhere in this book, including navigation controls, data-based list controls, and security controls. In this chapter, you'll focus on a few useful web controls that don't fit neatly into any of these categories. All of these controls appear in the Standard tab of the Visual Studio Toolbox.

The Calendar

The *Calendar control* presents a miniature calendar that you can place in any web page. Like most rich controls, the Calendar can be programmed as a single object (and defined in a single simple tag), but it renders itself with dozens of lines of HTML output.

```
<asp:Calendar id="MyCalendar" runat="server" />
```

The Calendar control presents a single-month view, as shown in Figure 10-1. The user can navigate from month to month by using the navigational arrows, at which point the page is posted back and ASP.NET automatically provides a new page with the correct month values. You don't need to write any additional event-handling code to manage this process. When the user clicks a date, the date becomes highlighted in a gray box (by default). You can retrieve the selected day in your code as a DateTime object from the Calendar.SelectedDate property.

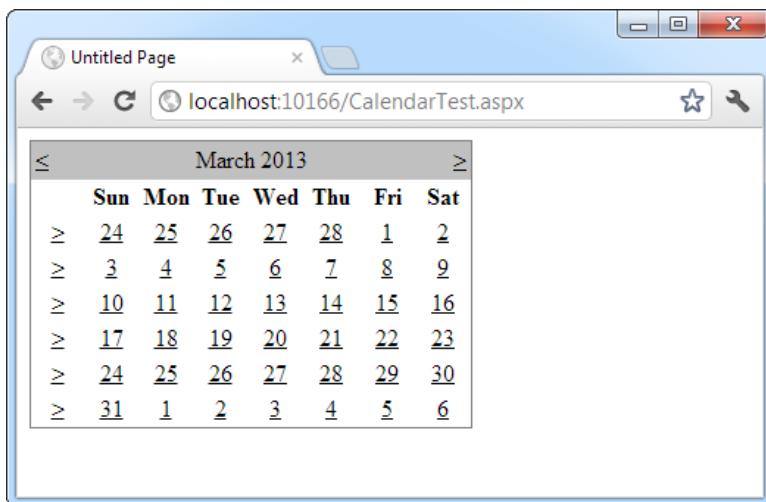


Figure 10-1. The default Calendar

This basic set of features may provide everything you need in your application. Alternatively, you can configure different selection modes through the `CalendarSelectionMode` property. Depending on the value you choose, you can allow users to select days (Day), entire weeks (DayWeek), whole months (DayWeekMonth), or render the control as a static calendar that doesn't allow selection (None). The only fact you must remember is that if you allow month selection, the user can also select a single week or a day. Similarly, if you allow week selection, the user can also select a single day.

You may also want to set the `Calendar.FirstDayOfWeek` property to configure how a week is shown. (For example, set `FirstDayOfWeek` to the enumerated value `Sunday`, and weeks will be selected from Sunday to Saturday.)

When you allow multiple date selection, you need to examine the `SelectedDates` property, which provides a collection of all the selected dates. You can loop through this collection by using the `foreach` syntax. The following code demonstrates this technique:

```
lblDates.Text = "You selected these dates:<br />";
foreach (DateTime dt in MyCalendar.SelectedDates)
{
    lblDates.Text += dt.ToString("MM/dd/yyyy") + "<br />";
}
```

Figure 10-2 shows the resulting page after this code has been executed.

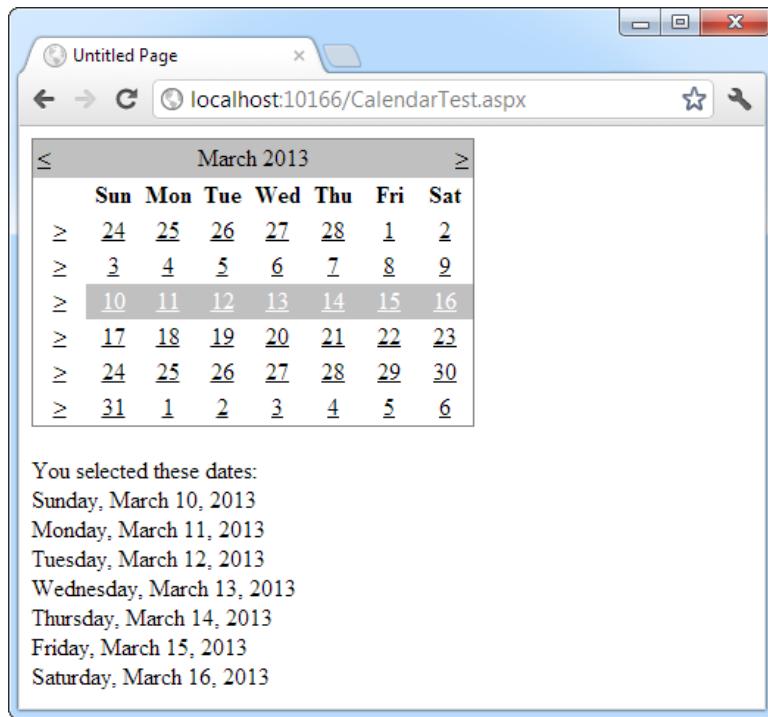


Figure 10-2. Selecting multiple dates

Formatting the Calendar

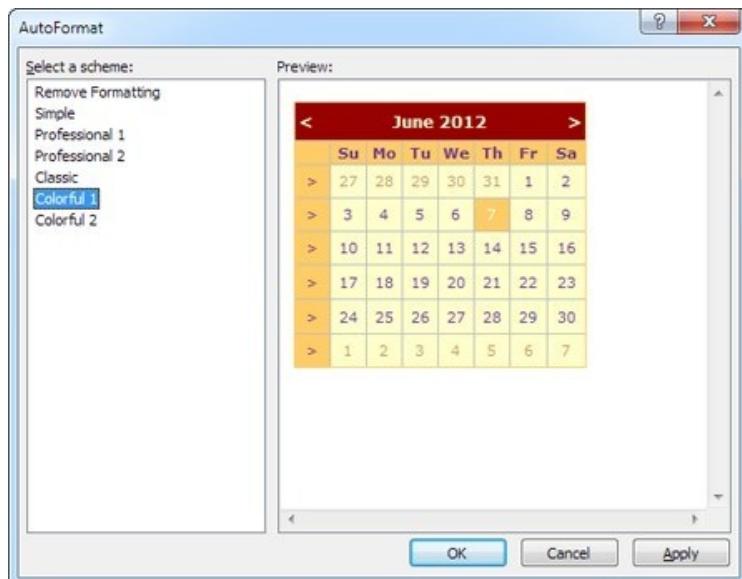
The Calendar control provides a whole host of formatting-related properties. You can set various parts of the calendar, such as the header, selector, and various day types, by using one of the style properties (for example, `WeekendDayStyle`). Each of these style properties references a full-featured `TableItemStyle` object that provides properties for coloring, border style, font, and alignment. Taken together, they allow you to modify almost any part of the calendar's appearance.

Table 10-1 lists the style properties that the Calendar control provides.

Table 10-1. Properties for Calendar Styles

Member	Description
DayHeaderStyle	The style for the section of the Calendar that displays the days of the week (as column headers).
DayStyle	The default style for the dates in the current month.
NextPrevStyle	The style for the navigation controls in the title section that move from month to month.
OtherMonthDayStyle	The style for the dates that aren't in the currently displayed month. These dates are used to "fill in" the calendar grid. For example, the first few cells in the topmost row may display the last few days from the previous month.
SelectedDayStyle	The style for the selected dates on the calendar.
SelectorStyle	The style for the week and month date selection controls.
TitleStyle	The style for the title section.
TodayDayStyle	The style for the date designated as today (represented by the TodaysDate property of the Calendar control).
WeekendDayStyle	The style for dates that fall on the weekend.

You can adjust each style by using the Properties window. For a quick shortcut, you can set an entire related color scheme by using the Calendar's Auto Format feature. To do so, start by selecting the Calendar on the design surface of a web form. Then click the arrow icon that appears next to its top-right corner to show the Calendar's smart tag, and click the Auto Format link. You'll be presented with a list of predefined formats that set the style properties, as shown in Figure 10-3.

**Figure 10-3.** Calendar styles

You can also use additional properties to hide some elements or configure the text they display. For example, properties that start with *Show* (such as ShowDayHeader, ShowTitle, and ShowGridLines) can be used to hide or show a specific visual element. Properties that end in *Text* (such as PrevMonthText, NextMonthText, and SelectWeekText) allow you to set the text that's shown in part of the calendar.

Restricting Dates

In most situations where you need to use a calendar for selection, you don't want to allow the user to select any date in the calendar. For example, the user might be booking an appointment or choosing a delivery date—two services that are generally provided only on set days. The Calendar control makes it surprisingly easy to implement this logic. In fact, if you've worked with the date and time controls on the Windows platform, you'll quickly recognize that the ASP.NET versions are far superior.

The basic approach to restricting dates is to write an event handler for the `Calendar.DayRender` event. This event occurs when the Calendar control is about to create a month to display to the user. This event gives you the chance to examine the date that is being added to the current month (through the `e.Day` property) and decide whether it should be selectable or restricted.

The following code makes it impossible to select any weekend days or days in years later than 2013:

```
protected void MyCalendar_DayRender(Object source, DayRenderEventArgs e)
{
    // Restrict dates after the year 2013 and those on the weekend.
    if (e.Day.IsWeekend || e.Day.Date.Year > 2013)
    {
        e.Day.IsSelectable = false;
    }
}
```

The `e.Day` object is an instance of the `CalendarDay` class, which provides various properties. Table 10-2 describes some of the most useful.

Table 10-2. *CalendarDay* Properties

Property	Description
Date	The <code>DateTime</code> object that represents this date.
IsWeekend	True if this date falls on a Saturday or Sunday.
IsToday	True if this value matches the <code>Calendar.TodaysDate</code> property, which is set to the current day by default.
IsOtherMonth	True if this date doesn't belong to the current month but is displayed to fill in the first or last row. For example, this might be the last day of the previous month or the next day of the following month.
IsSelectable	Allows you to configure whether the user can select this day.

The DayRender event is extremely powerful. Besides allowing you to tailor what dates are selectable, it also allows you to configure the cell where the date is located through the e.Cell property. (The calendar is displayed using an HTML table.) For example, you could highlight an important date or even add information. Here's an example that highlights a single day—the fifth of May—by adding a new Label control in the table cell for that day:

```
protected void MyCalendar_DayRender(Object source, DayRenderEventArgs e)
{
    // Check for May 5 in any year, and format it.
    if (e.Day.Date.Day == 5 && e.Day.Date.Month == 5)
    {
        e.Cell.BackColor = System.Drawing.Color.Yellow;

        // Add some static text to the cell.
        Label lbl = new Label();
        lbl.Text = "<br />My Birthday!";
        e.Cell.Controls.Add(lbl);
    }
}
```

Figure 10-4 shows the resulting calendar display.

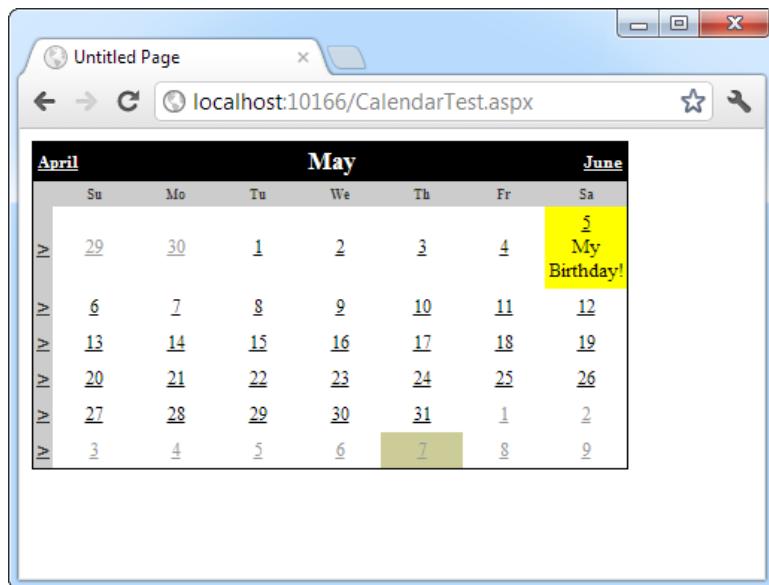


Figure 10-4. Highlighting a day

The Calendar control provides two other useful events: SelectionChanged and VisibleMonthChanged. These occur immediately after the user selects a new day or browses to a new month (using the next month and previous month links). You can react to these events and update other portions of the web page to correspond to the current calendar month. For example, you could design a page that lets you schedule a meeting in two steps. First you choose the appropriate day. Then you choose one of the available times on that day.

The following code demonstrates this approach, using a different set of time values if a Monday is selected in the calendar than it does for other days:

```
protected void MyCalendar_SelectionChanged(Object source, EventArgs e)
{
    lstTimes.Items.Clear();

    switch (MyCalendar.SelectedDate.DayOfWeek)
    {
        case DayOfWeek.Monday:
            // Apply special Monday schedule.
            lstTimes.Items.Add("10:00");
            lstTimes.Items.Add("10:30");
            lstTimes.Items.Add("11:00");
            break;
        default:
            lstTimes.Items.Add("10:00");
            lstTimes.Items.Add("10:30");
            lstTimes.Items.Add("11:00");
            lstTimes.Items.Add("11:30");
            lstTimes.Items.Add("12:00");
            lstTimes.Items.Add("12:30");
            break;
    }
}
```

To try these features of the Calendar control, run the Appointment.aspx page from the online samples. This page provides a formatted Calendar control that restricts some dates, formats others specially, and updates a corresponding list control when the selection changes.

Table 10-3 gives you an at-a-glance look at almost all the members of the Calendar control class.

Table 10-3. Calendar Members

Member	Description
Caption and CaptionAlign	Gives you an easy way to add a title to the calendar. By default, the caption appears at the top of the title area, just above the month heading. However, you can control this to some extent with the CaptionAlign property. Use Left or Right to keep the caption at the top but move it to one side or the other, and use Bottom to place the caption under the calendar.
CellPadding	ASP.NET creates a date in a separate cell of an invisible table. CellPadding is the space, in pixels, between the border of each cell and its contents.
CellSpacing	The space, in pixels, between cells in the same table.
DayNameFormat	Determines how days are displayed in the calendar header. Valid values are Full (as in Sunday), FirstLetter (S), FirstTwoLetters (Su), and Short (Sun), which is the default.
FirstDayOfWeek	Determines which day is displayed in the first column of the calendar. The values are any day name from the FirstDayOfWeek enumeration (such as Sunday). By default, this is Sunday.
NextMonthText and PrevMonthText	Sets the text that the user clicks to move to the next or previous month. These navigation links appear at the top of the calendar and are the greater-than (>) and less-than (<) signs by default. This setting is applied only if NextPrevFormat is set to CustomText.
NextPrevFormat	Sets the text that the user clicks to move to the next or previous month. This can be FullMonth (for example, December), ShortMonth (Dec), or CustomText, in which case the NextMonthText and PrevMonthText properties are used. CustomText is the default.
SelectedDate and SelectedDates	Sets or gets the currently selected date as a DateTime object. You can specify this in the control tag in a format like 12:00:00 AM, 12/31/2010 (depending on your computer's regional settings). If you allow multiple date selection, the SelectedDates property will return a collection of DateTime objects, one for each selected date. You can use collection methods such as Add, Remove, and Clear to change the selection.
SelectionMode	Determines how many dates can be selected at once. The default is Day, which allows one date to be selected. Other options include DayWeek (a single date or an entire week) or DayWeekMonth (a single date, entire week, or entire month). You have no way to allow the user to select multiple noncontiguous dates. You also have no way to allow larger selections without also including smaller selections. (For example, if you allow full months to be selected, you must also allow week selection and individual day selection.)
SelectMonthText and SelectWeekText	The text shown for the link that allows the user to select an entire month or week. These properties don't apply if the SelectionMode is Day.

(continued)

Table 10-3. (continued)

Member	Description
ShowDayHeader, ShowGridLines, ShowNextPrevMonth and ShowTitle	These Boolean properties allow you to configure whether various parts of the calendar are shown, including the day titles, gridlines between every day, the previous/next month navigation links, and the title section. Note that hiding the title section also hides the next and previous month navigation controls.
TitleFormat	Configures how the month is displayed in the title area. Valid values include Month and MonthYear (the default).
TodaysDate	Sets which day should be recognized as the current date and formatted with the TodayDayStyle. This defaults to the current day on the web server.
VisibleDate	Gets or sets the date that specifies what month will be displayed in the calendar. This allows you to change the calendar display without modifying the current date selection.
DayRender event	Occurs once for each day that is created and added to the currently visible month before the page is rendered. This event gives you the opportunity to apply special formatting, add content, or restrict selection for an individual date cell. Keep in mind that days can appear in the calendar even when they don't fall in the current month, provided they fall close to the end of the previous month or close to the start of the following month.
SelectionChanged event	Occurs when the user selects a day, a week, or an entire month by clicking the date selector controls.
VisibleMonthChanged event	Occurs when the user clicks the next or previous month navigation controls to move to another month.

The AdRotator

The basic purpose of the *AdRotator* is to provide a graphic on a page that is chosen randomly from a group of possible images. In other words, every time the page is requested, an image is selected at random and displayed, which is the *rotation* indicated by the name *AdRotator*. One use of the *AdRotator* is to show banner-style advertisements on a page, but you can use it anytime you want to vary an image randomly.

Using ASP.NET, it wouldn't be too difficult to implement an *AdRotator* type of design on your own. You could react to the *Page.Load* event, generate a random number, and then use that number to choose from a list of predetermined image files. You could even store the list in the *web.config* file so that it can be easily modified separately as part of the application's configuration. Of course, if you wanted to enable several pages with a random image, you would either have to repeat the code or create your own custom control. The *AdRotator* provides these features for free.

ShOULD YOU ChOOSe USe the aDrOtatOr FOr WeB aDVertISING?

The name of the AdRotator control is slightly misleading. A better name might be RandomImage or RotatingImage. That's because there's absolutely no requirement for the AdRotator content to have anything to do with advertising.

In fact, these days the AdRotator is not the most common way to deal with website advertising, even in an ASP.NET application. You're more likely to use a block of pregenerated JavaScript that's supplied to you by an advertising service. For example, if you decide to make some extra money showing ads with Google AdSense (www.google.com/adsense), Google will provide you with a block of JavaScript code that fetches an appropriate ad (or a combination of ad links) for your page. The ads Google returns are random, but based on a variety of details, including the content it detects on your page and the amount of space you've allocated for advertising. Other advertising approaches are more sophisticated and use pop-up panels, Flash animation, among other tricks.

The Advertisement File

The AdRotator stores its list of image files in an XML file. This file uses the format shown here:

```
<Advertisements>
<Ad>
  <ImageUrl>prosetech.jpg</ImageUrl>
  <NavigateUrl>http://www.prosetech.com</NavigateUrl>
  <AlternateText>ProseTech Site</AlternateText>
  <Impressions>1</Impressions>
  <Keyword>Computer</Keyword>
</Ad>
</Advertisements>
```

Tip An XML file is just a text file with specific tags. You can create an XML file by using nothing more than a text editor such as Notepad, but you can also use the Visual Studio text editor. Just choose Website Item from the menu, and then choose XML File. It's up to you to fill in the right tags and content. You can save the advertisements file wherever you'd like—either in the main website folder or in a subfolder that you've created.

This example shows a single possible advertisement, which the AdRotator control picks at random from the list of advertisements. To add more advertisements, you would create multiple `<Ad>` elements and place them all inside the root `<Advertisements>` element:

```
<Advertisements>
<Ad>
  <!-- First ad here. -->
</Ad>

<Ad>
  <!-- Second ad here. -->
</Ad>
</Advertisements>
```

Each <Ad> element has a number of other important properties that configure the link, the image, and the frequency, as described in Table 10-4.

Table 10-4. Advertisement File Elements

Element	Description
ImageUrl	The image that will be displayed. This can be a relative link (a file in the current directory) or a fully qualified Internet URL.
NavigateUrl	The link that will be followed if the user clicks the banner. This can be a relative or fully qualified URL.
AlternateText	The text that will be displayed instead of the picture if it cannot be displayed. This text will also be used as a tooltip in some newer browsers.
Impressions	A number that sets how often an advertisement will appear. This number is relative to the numbers specified for other ads. For example, a banner with the value 10 will be shown twice as often (on average) as the banner with the value 5.
Keyword	A keyword that identifies a group of advertisements. You can use this for filtering. For example, you could create ten advertisements and give half of them the keyword Retail and the other half the keyword Computer. The web page can then choose to filter the possible advertisements to include only one of these groups.

The AdRotator Class

The actual AdRotator class provides a limited set of properties. You specify both the appropriate advertisement file in the AdvertisementFile property and the type of window that the link should follow (the Target window). The target can name a specific frame, or it can use one of the values defined in Table 10-5.

Table 10-5. Special Frame Targets

Target	Description
_blank	The link opens a new unframed window.
_parent	The link opens in the parent of the current frame.
_self	The link opens in the current frame.
_top	The link opens in the topmost frame of the current window (so the link appears in the full window).

Optionally, you can set the KeywordFilter property so that the banner will be chosen from a specific keyword group. This is a fully configured AdRotator tag:

```
<asp:AdRotator ID="Ads" runat="server" AdvertisementFile="MainAds.xml"
    Target="_blank" KeywordFilter="Computer" />
```

Additionally, you can react to the AdRotator.AdCreated event. This occurs when the page is being created and an image is randomly chosen from the advertisements file. This event provides you with information about the image that you can use to customize the rest of your page. For example, you might display some related content or a link, as shown in Figure 10-5.

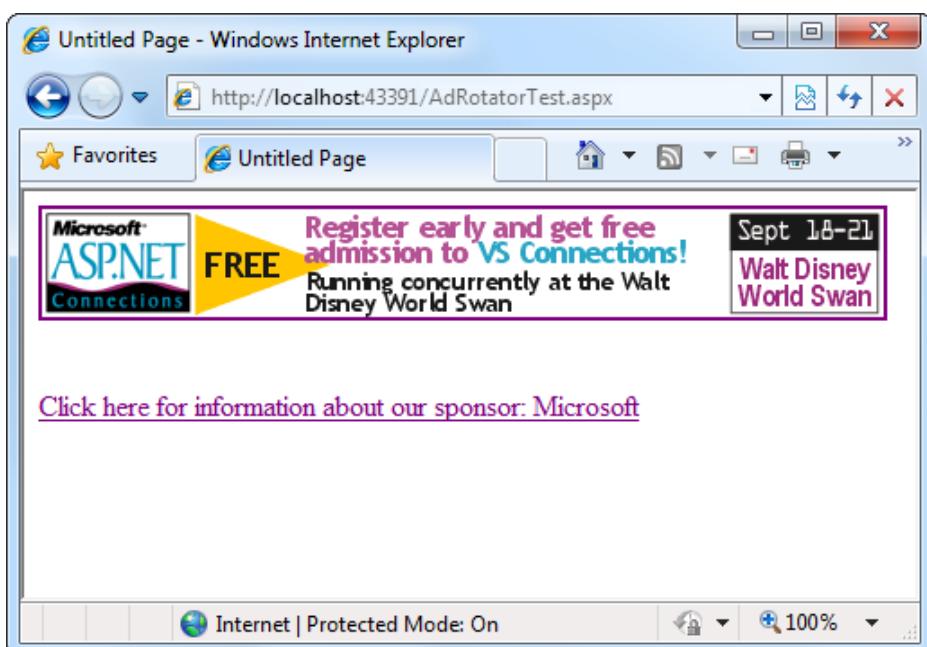


Figure 10-5. An AdRotator with synchronized content

The event-handling code for this example simply configures a HyperLink control named InkBanner based on the randomly selected advertisement:

```
protected void Ads_AdCreated(Object sender, AdCreatedEventArgs e)
{
    // Synchronize the Hyperlink control.
    InkBanner.NavigateUrl = e.NavigateUrl;

    // Syncrhonize the text of the link.
    InkBanner.Text = "Click here for information about our sponsor: ";
    InkBanner.Text += e.AlternateText;
}
```

As you can see, rich controls such as the Calendar and AdRotator don't just add a sophisticated HTML output; they also include an event framework that allows you to take charge of the control's behavior and integrate it into your application.

Pages with Multiple Views

In a typical website, you'll surf through many separate pages. For example, if you want to add an item to your shopping cart and take it to the checkout in an e-commerce site, you'll need to jump from one page to another. This design has its advantages—namely, it lets you carefully separate different tasks into different code files. It also presents some challenges; for example, you need to come up with a way to transfer information from one page to another (as you saw in Chapter 8).

However, in some cases it makes more sense to create a single page that can handle several different tasks. For example, you might want to provide several views of the same data (such as a grid-based view and a chart-based view) and allow the user to switch from one view to the other without leaving the page. Or, you might want to handle a small multistep task in one place (such as supplying user information for an account sign-up process). In these examples, you need a way to create dynamic pages that provide more than one possible view. Essentially, the page hides and shows different controls depending on which view you want to present.

The simplest way to understand this technique is to create a page with several Panel controls. Each panel can hold a group of ASP.NET controls. For example, imagine you're creating a simple three-step wizard. You'll start by adding three panels to your page, one for each step—say, panelStep1, panelStep2, and panelStep3. You can place the panels one after the other, because you'll show only one at a time. After you've added the panels, you can place the appropriate controls inside each panel. To start, the Visible property of each panel should be false, except for panelStep1, which appears the first time the user requests the page.

Here's an example that shows the way you can arrange your panels:

```
<asp:Panel ID="panelStep1" runat="server">...</asp:Panel>
<asp:Panel ID="panelStep2" Visible="False" runat="server">...</asp:Panel>
<asp:Panel ID="panelStep3" Visible="False" runat="server">...</asp:Panel>
```

Note When you set the Visible property of a control to false, the control won't appear in the page at render time. Any controls inside an invisible panel are also hidden from sight, and they won't be present in the rendered page. However, these controls will still appear in the Visual Studio design surface so that you can still see and configure them.

Finally, you'll add one or more navigation buttons outside the panels. For example, the following code handles the click of a Next button, which is placed just after panelStep3 (so it always appears at the bottom of the page). The code checks which step the user is currently on, hides the current panel, and shows the following panel. This way, the user is moved to the next step.

```
protected void cmdNext_Click(object sender, EventArgs e)
{
    if (panelStep1.Visible)
    {
        // Move to step 2.
        panelStep1.Visible = false;
        panelStep2.Visible = true;
    }
    else if (panelStep2.Visible)
    {
        // Move to step 3.
        panelStep2.Visible = false;
        panelStep3.Visible = true;

        // Change text of button from Next to Finish.
        cmdNext.Text = "Finish";
    }
    else if (panelStep3.Visible)
    {
        // The wizard is finished.
        panelStep3.Visible = false;
    }
}
```

```

    // Add code here to perform the appropriate task
    // with the information you've collected.
}
}

```

This approach works relatively well. Even when the panels are hidden, you can still interact with all the controls on each panel and retrieve the information they contain. The problem is that you need to write all the code for controlling which panel is visible. If you make your wizard much more complex—for example, you want to add a button for returning to a previous step—it becomes more difficult to keep track of what's happening. At best, this approach clutters your page with the code for managing the panels. At worst, you'll make a minor mistake and end up with two panels showing at the same time.

Fortunately, ASP.NET gives you a more robust option. You can use two controls that are designed for the job—the MultiView and the Wizard. In the following sections, you'll see how you can use both of these controls with the GreetingCardMaker example developed in Chapter 6.

The MultiView Control

The MultiView is the simpler of the two multiple-view controls. Essentially, the MultiView gives you a way to declare multiple views and show only one at a time. It has no default user interface—you get only whatever HTML and controls you add. The MultiView is equivalent to the custom panel approach explained earlier.

Creating a MultiView is suitably straightforward. You add the `<asp:MultiView>` tag to your .aspx page file and then add one `<asp:View>` tag inside it for each separate view:

```

<asp:MultiView ID="MultiView1" runat="server">
    <asp:View ID="View1" runat="server">...</asp:View>
    <asp:View ID="View2" runat="server">...</asp:View>
    <asp:View ID="View3" runat="server">...</asp:View>
</asp:MultiView>

```

In Visual Studio, you create these tags by first dropping a MultiView control onto your form and then using the Toolbox to add as many View controls inside it as you want. This drag-and-drop process can be a bit tricky. When you add the first View control, you must make sure to drop it in the blank area inside the MultiView (not next to the MultiView, or on the MultiView's title bar). When you add more View controls, you must drop each one on one of the gray header bars of one of the existing views. The gray header has the View title (such as View1 or View2).

The View control plays the same role as the Panel control in the previous example, and the MultiView takes care of coordinating all the views so that only one is visible at a time.

Inside each view, you can add HTML or web controls. For example, consider the GreetingCardMaker example demonstrated in Chapter 6, which allows the user to create a greeting card by supplying some text and choosing colors, a font, and a background. As the GreetingCardMaker grows more complex, it requires more controls, and it becomes increasingly difficult to fit all those controls on the same page. One possible solution is to divide these controls into logical groups and place each group in a separate view.

Creating Views

Here's the full markup for a MultiView that splits the greeting card controls into three views named View1, View2, and View3:

```

<asp:MultiView ID="MultiView1" runat="server" >

    <asp:View ID="View1" runat="server">
        Choose a foreground (text) color:<br />
    
```

```
<asp:DropDownList ID="lstForeColor" runat="server" AutoPostBack="True"
    OnSelectedIndexChanged="ControlChanged" />
<br /><br />
Choose a background color:<br />
<asp:DropDownList ID="lstBackColor" runat="server" AutoPostBack="True"
    OnSelectedIndexChanged="ControlChanged" />
</asp:View>

<asp:View ID="View2" runat="server">
    Choose a border style:<br />
    <asp:RadioButtonList ID="lstBorder" runat="server" AutoPostBack="True"
        OnSelectedIndexChanged="ControlChanged" RepeatColumns="2" />
    <br />
    <asp:CheckBox ID="chkPicture" runat="server" AutoPostBack="True"
        OnCheckedChanged="ControlChanged" Text="Add the Default Picture" />
</asp:View>

<asp:View ID="View3" runat="server">
    Choose a font name:<br />
    <asp:DropDownList ID="lstFontName" runat="server" AutoPostBack="True"
        OnSelectedIndexChanged="ControlChanged" />
    <br /><br />
    Specify a font size:<br />
    <asp:TextBox ID="txtFontSize" runat="server" AutoPostBack="True"
        OnTextChanged="ControlChanged" />
    <br /><br />
    Enter the greeting text below:<br />
    <asp:TextBox ID="txtGreeting" runat="server" AutoPostBack="True"
        OnTextChanged="ControlChanged" TextMode="MultiLine" />
</asp:View>

</asp:MultiView>
```

Visual Studio shows all your views at design time, one after the other (see Figure 10-6). You can edit these regions in the same way you design any other part of the page.

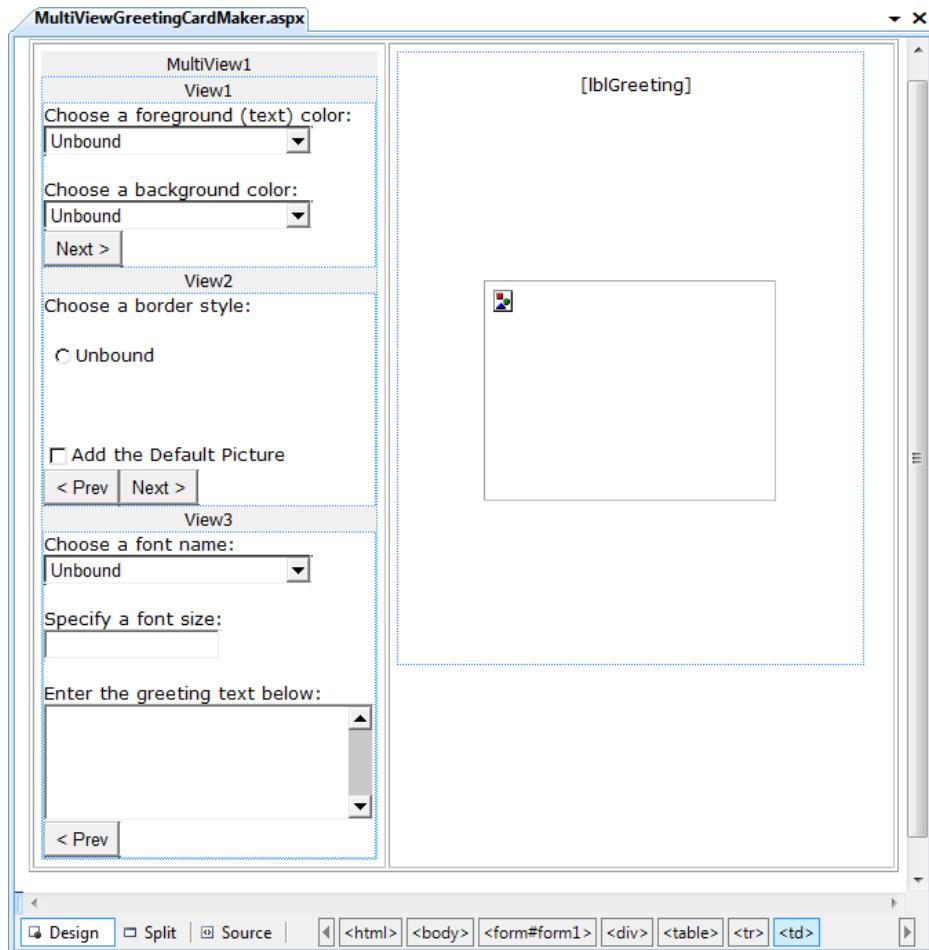


Figure 10-6. Designing multiple views

Showing a View

If you run this example, you won't see what you expect. The MultiView will appear empty on the page, and all the controls in all your views will be hidden.

The reason this happens is that the MultiView.ActiveViewIndex property is, by default, set to -1. The ActiveViewIndex property determines which view will be shown. If you set the ActiveViewIndex to 0, however, you'll see the first view. Similarly, you can set it to 1 to show the second view, and so on. You can set this property by using the Properties window or using code:

```
// Show the first view.
MultiView1.ActiveViewIndex = 0;
```

This example shows the first view (View1) and hides whatever view is currently being displayed, if any.

Tip To make more-readable code, you can create an enumeration that defines a name for each view. You can set the ActiveViewIndex by using the descriptive name from the enumeration rather than an order. Refer to Chapter 3 for a refresher on enumerations.

You can also use the SetActiveView() method, which accepts any one of the view objects you've created, rather than the view name. By using the view objects, you can catch errors earlier. For example, if you misspell a view name, Visual Studio will spot the problem when you compile your code, rather than allowing your code to fail at runtime when it attempts to load the view.

```
MultiView1.SetActiveView(View1);
```

This gives you enough functionality that you can create previous and next navigation buttons. However, it's still up to you to write the code that checks which view is visible and changes the view. This code is a little simpler, because you don't need to worry about hiding views any longer, but it's still less than ideal.

Fortunately, the MultiView includes some built-in smarts that can save you a lot of trouble. Here's how it works: the MultiView recognizes button controls with specific command names. (Technically, a button control is any control that implements the IButtonControl interface, including the Button, ImageButton, and LinkButton.) If you add a button control to the view that uses one of these recognized command names, the button gets some automatic functionality. Using this technique, you can create navigation buttons without writing any code.

Table 10-6 lists all the recognized command names. Each command name also has a corresponding static field in the MultiView class, so you can easily get the right command name if you choose to set it programmatically.

Table 10-6. Recognized Command Names for the MultiView

Command Name	MultiView Field	Description
PrevView	PreviousViewCommandName	Moves to the previous view.
NextView	NextViewCommandName	Moves to the next view.
SwitchViewByID	SwitchViewByIDCommandName	Moves to the view with a specific ID (string name). The ID is taken from the CommandArgument property of the button control.
SwitchViewByIndex	SwitchViewByIndexCommandName	Moves to the view with a specific numeric index. The index is taken from the CommandArgument property of the button control.

To try this, add this button to the first view:

```
<asp:Button ID="Button1" runat="server" CommandArgument="View2"
CommandName="SwitchViewByID" Text="Go to View2" />
```

When clicked, this button sets the MultiView to show the view specified by the CommandArgument (View2).

Rather than create buttons that take the user to a specific view, you might want a button that moves forward or backward one view. To do this, you use the PrevView and NextView command names. Here's an example that defines previous and next buttons in the second View:

```
<asp:Button ID="Button1" runat="server" Text="< Prev" CommandName="PrevView" />
<asp:Button ID="Button2" runat="server" Text="Next >" CommandName="NextView" />
```

After you add these buttons to your view, you can move from view to view easily. Figure 10-7 shows the previous example with the second view currently visible.

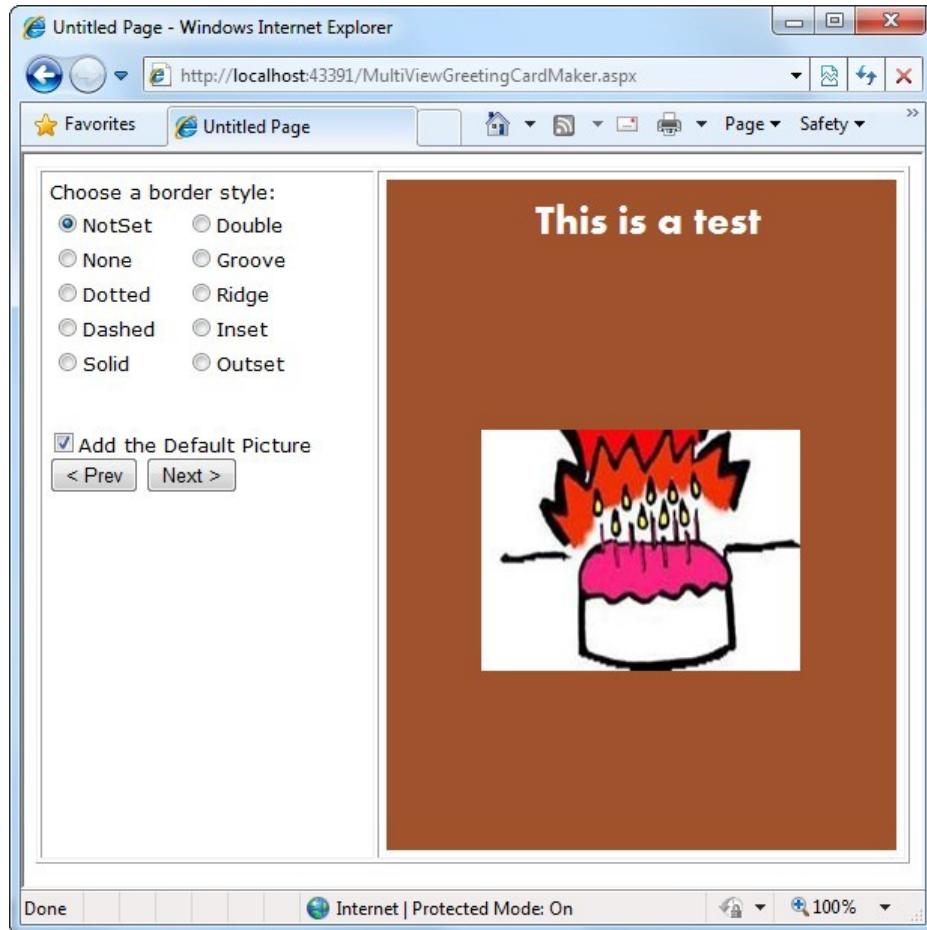


Figure 10-7. Moving from one view to another

Tip Be careful how many views you cram into a single page. When you use the MultiView control, the control model—including the controls from every view—is created on every postback and persisted to view state. In most situations, this won't be a significant factor. However, it increases the overall page size, especially when tweaking controls programmatically (which increases the amount of information they need to store in view state).

The Wizard Control

The Wizard control is a more glamorous version of the MultiView control. It also supports showing one of several views at a time, but it includes a fair bit of built-in yet customizable behavior, including navigation buttons, a sidebar with step links, styles, and templates.

Usually, wizards represent a single task, and the user moves linearly through them, moving from the current step to the one immediately following it (or the one immediately preceding it in the case of a correction). The ASP.NET Wizard control also supports nonlinear navigation, which means it allows you to decide to ignore a step based on the information the user supplies.

By default, the Wizard control supplies navigation buttons and a sidebar with links for each step on the left. You can hide the sidebar by setting the `Wizard.DisplaySideBar` property to false. Usually, you'll take this step if you want to enforce strict step-by-step navigation and prevent the user from jumping out of sequence. You supply the content for each step by using any HTML or ASP.NET controls. Figure 10-8 shows the region where you can add content to an out-of-the-box Wizard instance.

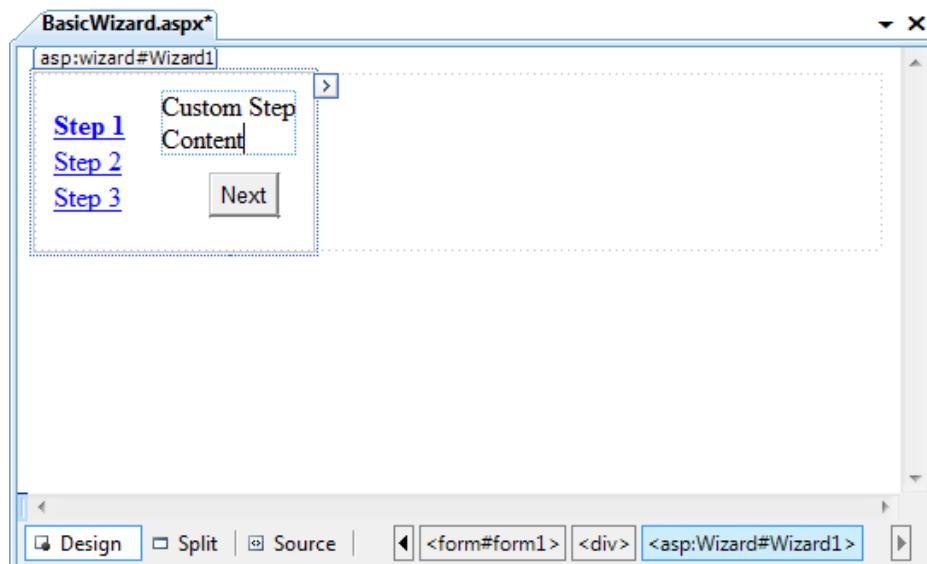


Figure 10-8. The region for step content

Defining Wizard Steps

To create a wizard in ASP.NET, you simply define the steps and their content by using `<asp:WizardStep>` tags. Here's the basic structure you'll use:

```
<asp:Wizard ID="Wizard1" runat="server" ... >
<WizardSteps>

  <asp:WizardStep runat="server" Title="Step 1">
    ...
  </asp:WizardStep>
```

```
<asp:WizardStep runat="server" Title="Step 2">
...
</asp:WizardStep>

...
<WizardSteps>
</asp:Wizard>
```

You can add as many WizardStep controls inside the Wizard as you want. Conceptually, the WizardStep plays the same role as the View in a MultiView (or the basic Panel in the first example that you considered). You place the content for each step inside the WizardStep control.

Before you start adding the content to your wizard, it's worth reviewing Table 10-7, which shows a few basic pieces of information that you can define for each step.

Table 10-7. WizardStep Properties

Property	Description
Title	The descriptive name of the step. This name is used for the text of the links in the sidebar.
StepType	The type of step, as a value from the WizardStepType enumeration. This value determines the type of navigation buttons that will be shown for this step. Choices include Start (shows a Next button), Step (shows Next and Previous buttons), Finish (shows Finish and Previous buttons), Complete (shows no buttons and hides the sidebar, if it's enabled), and Auto (the step type is inferred from the position in the collection). The default is Auto, which means the first step is Start, the last step is Finish, and all other steps are Step.
AllowReturn	Indicates whether the user can return to this step. If false, the user will not be able to return after passing this step. The sidebar link for this step will have no effect, and the Previous button of the following step will either skip this step or be hidden completely (depending on the AllowReturn value of the preceding steps).

To see how this works, consider a wizard that again uses the GreetingCardMaker example. It guides the user through four steps. The first three steps allow the user to configure the greeting card, and the final step shows the generated card.

```
<asp:Wizard ID="Wizard1" runat="server" ActiveStepIndex="0"
BackColor="LemonChiffon" BorderStyle="Groove" BorderWidth="2px" CellPadding="10">

<WizardSteps>
<asp:WizardStep runat="server" Title="Step 1 - Colors">
    Choose a foreground (text) color:<br />
    <asp:DropDownList ID="lstForeColor" runat="server" />
    <br />
    Choose a background color:<br />
    <asp:DropDownList ID="lstBackColor" runat="server" />
</asp:WizardStep>

<asp:WizardStep runat="server" Title="Step 2 - Background">
    Choose a border style:<br />
```

```

<asp:RadioButtonList ID="lstBorder" runat="server" RepeatColumns="2" />
<br /><br />
<asp:CheckBox ID="chkPicture" runat="server"
  Text="Add the Default Picture" />
</asp:WizardStep>

<asp:WizardStep runat="server" Title="Step 3 - Text">
  Choose a font name:<br />
  <asp:DropDownList ID="lstFontName" runat="server" />
  <br /><br />
  Specify a font size:<br />
  <asp:TextBox ID="txtFontSize" runat="server" />
  <br /><br />
  Enter the greeting text below:<br />
  <asp:TextBox ID="txtGreeting" runat="server"
    TextMode="MultiLine" />
</asp:WizardStep>

<asp:WizardStep runat="server" StepType="Complete" Title="Greeting Card">
  <asp:Panel ID="pnlCard" runat="server" HorizontalAlign="Center">
    <br />
    <asp:Label ID="lblGreeting" runat="server" />
    <asp:Image ID="imgDefault" runat="server" Visible="False" />
  </asp:Panel>
</asp:WizardStep>
</WizardSteps>

</asp:Wizard>

```

If you look carefully, you'll find a few differences from the original page and the MultiView-based example. First, the controls aren't set to automatically post back. That's because the greeting card isn't rendered until the final step, at the conclusion of the wizard. (You'll learn more about how to handle this event in the next section.) Another change is that no navigation buttons exist. That's because the wizard adds these details automatically based on the step type. For example, you'll get a Next button for the first two steps, a Previous button for steps 2 and 3, and a Finish button for step 3. The final step, which shows the complete card, doesn't provide any navigation links because the StepType is set to Complete. Figure 10-9 shows the wizard steps.

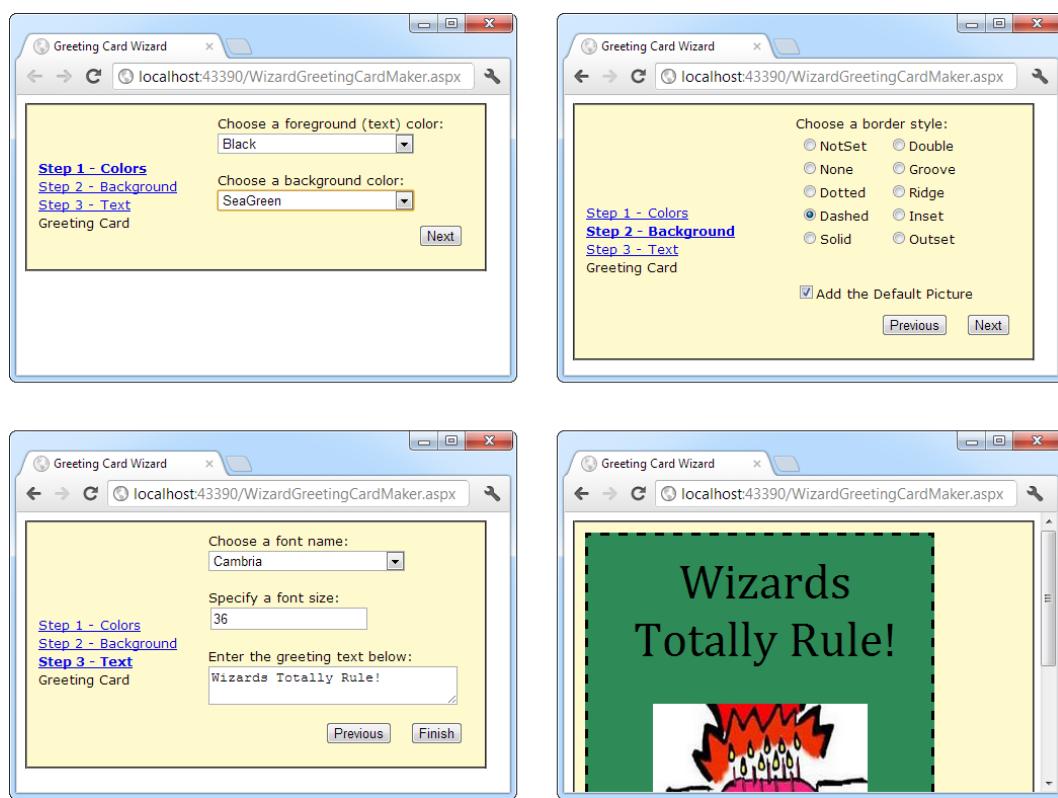


Figure 10-9. A wizard with four steps

Unlike the MultiView control, you can see only one step at a time in Visual Studio. To choose which step you're currently designing, select it from the smart tag, as shown in Figure 10-10. But be warned—every time you do, Visual Studio changes the Wizard.ActiveStepIndex property to the step you choose. Make sure you set this back to 0 before you run your application so it starts at the first step.

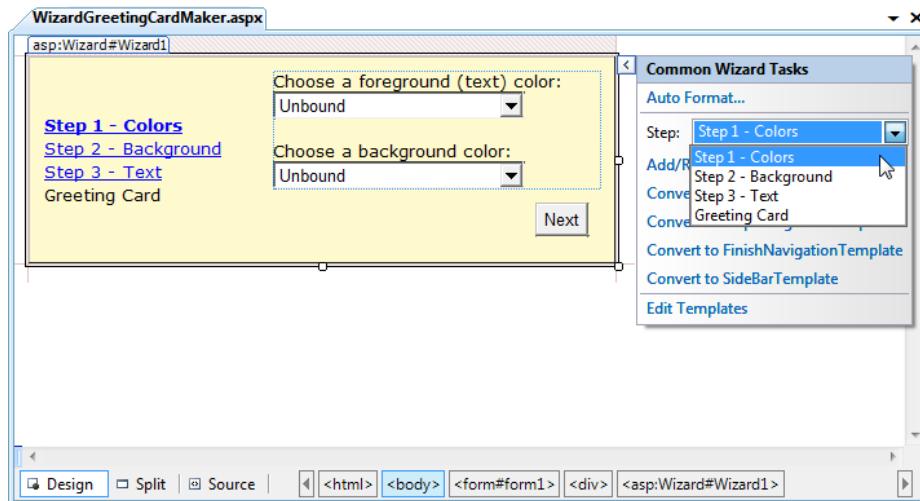


Figure 10-10. Designing a step

Note Remember, when you add controls to separate steps on a wizard, the controls are all instantiated in view state, regardless of which step is currently shown. If you need to slim down a complex wizard, you may need to split it into separate pages, use the Server.Transfer() method to move from one page to the next, or use a less elegant programming model.

Using Wizard Events

You can write the code that underpins your wizard by responding to several events (as listed in Table 10-8).

Table 10-8. Wizard Events

Event	Description
ActiveStepChanged	Occurs when the control switches to a new step (either because the user has clicked a navigation button or because your code has changed the ActiveStepIndex property).
CancelButtonClick	Occurs when the Cancel button is clicked. The Cancel button is not shown by default, but you can add it to every step by setting the Wizard.DisplayCancelButton property. Usually, a Cancel button exits the wizard. If you don't have any cleanup code to perform, just set the CancelDestinationPageUrl property, and the wizard will take care of the redirection automatically.
FinishButtonClick	Occurs when the Finish button is clicked.
NextButtonClick and PreviousButtonClick	Occurs when the Next or Previous button is clicked in any step. However, because there is more than one way to move from one step to the next, it's often easier to handle the ActiveStepChanged event.
SideBarButtonClick	Occurs when a button in the sidebar area is clicked.

On the whole, two wizard programming models exist:

Commit-as-you-go: This makes sense if each wizard step wraps an atomic operation that can't be reversed. For example, if you're processing an order that involves a credit card authorization followed by a final purchase, you probably don't want the user to step back and edit the credit card number after it's been authorized. To support the commit-as-you-go model, you set the AllowReturn property to false on some or all steps. You may also want to respond to the ActiveStepChanged event to commit changes for each step.

Commit-at-the-end: This makes sense if each wizard step is collecting information for an operation that's performed only at the end. For example, if you're collecting user information and plan to generate a new account after you have all the information, you'll probably allow a user to make changes midway through the process. You execute your code for generating the new account when the wizard ends by reacting to the FinishButtonClick event.

To implement commit-at-the-end with the current example, just respond to the FinishButtonClick event. For example, to implement the greeting card wizard, you simply need to respond to this event and call UpdateCard(), the private method that refreshes the greeting card:

```
protected void Wizard1_FinishButtonClick(object sender,
    WizardNavigationEventArgs e)
{
    UpdateCard();
}
```

For the complete code for the UpdateCard() method, which generates the greeting card, refer to Chapter 6 (or check out the downloadable sample code).

If you decide to use the commit-as-you go model, you would respond to the ActiveStepChanged event and call UpdateCard() at that point to refresh the card every time the user moves from one step to another. This assumes the greeting card is always visible. (In other words, it's not contained in the final step of the wizard.) The commit-as-you-go model is similar to the previous example that used the MultiView.

Formatting the Wizard

Without a doubt, the Wizard control's greatest strength is the way it lets you customize its appearance. This means if you want the basic model (a multistep process with navigation buttons and various events), you aren't locked into the default user interface.

Depending on how radically you want to change the wizard, you have several options. For less-dramatic modifications, you can set various top-level properties of the Wizard control. For example, you can control the colors, fonts, spacing, and border style, as you can with any ASP.NET control. You can also tweak the appearance of every button. For example, to change the Next button, you can use the following properties: StepNextButtonType (use a button, link, or clickable image), StepNextButtonText (customize the text for a button or link), StepNextButtonImageUrl (set the image for an image button), and StepNextButtonStyle (use a style from a style sheet). You can also add a header by using the HeaderText property.

More control is available through styles. You can use styles to apply formatting options to various portions of the Wizard control just as you can use styles to format parts of rich data controls such as the GridView. Table 10-9 lists all the styles you can use. As with other style-based controls, more-specific style settings (such as SideBarStyle) override more-general style settings (such as ControlStyle) when they conflict. Similarly, StartNextButtonStyle overrides NavigationButtonStyle on the first step.

Table 10-9. Wizard Styles

Style	Description
ControlStyle	Applies to all sections of the Wizard control.
HeaderStyle	Applies to the header section of the wizard, which is visible only if you set some text in the HeaderText property.
BorderStyle	Applies to the border around the Wizard control. You can use it in conjunction with the BorderColor and BorderWidth properties.
SideBarStyle	Applies to the sidebar area of the wizard.
SideBarButtonStyle	Applies to just the buttons in the sidebar.
StepStyle	Applies to the section of the control where you define the step content.
NavigationStyle	Applies to the bottom area of the control where the navigation buttons are displayed.
NavigationButtonStyle	Applies to just the navigation buttons in the navigation area.
StartNextButtonStyle	Applies to the Next navigation button on the first step (when StepType is Start).
StepNextButtonStyle	Applies to the Next navigation button on intermediate steps (when StepType is Step).
StepPreviousButtonStyle	Applies to the Previous navigation button on intermediate steps (when StepType is Step).
FinishPreviousButtonStyle	Applies to the Previous navigation button on the last step (when StepType is Finish).
FinishCompleteButtonStyle	Applies to the Complete navigation button on the last step (when StepType is Finish).
CancelButtonStyle	Applies to the Cancel button, if you have Wizard.DisplayCancelButton set to true.

Note The Wizard control also supports templates, which give you a more radical approach to formatting. If you can't get the level of customization you want through properties and styles, you can use templates to completely define the appearance of each section of the Wizard control, including the headers and navigation links. Templates require data-binding expressions and are discussed in Chapter 15 and Chapter 16.

Validating with the Wizard

The FinishButtonClick, NextButtonClick, PreviousButtonClick, and SideBarButtonClick events are cancellable. That means that you can use code like this to prevent the requested navigation action from taking place:

```
protected void Wizard1_NextButtonClick(object sender,
    WizardNavigationEventArgs e)
```

```
{  
    // Perform some sort of check.  
    if (e.NextStepIndex == 1 && txtName.Text == "")  
    {  
        // Cancel navigation and display a message elsewhere on the page.  
        e.Cancel = true;  
        lblInfo.Text =  
            "You cannot move to the next step until you supply your name.";  
    }  
}
```

Here the code checks whether the user is trying to move to step 1 by using the `NextStepIndex` property. (Alternatively, you could examine the current step by using the `CurrentStepIndex` property.) If so, the code then checks a text box and cancels the navigation if it doesn't contain any text, keeping the user on the current step. Writing this sort of logic gets a little tricky, because you need to keep in mind that step-to-step navigation can be performed in several ways. To simplify your life, you can write one event handler that deals with the `NextButtonClick`, `PreviousButtonClick`, and `SideBarButtonClick` events, and performs the same check. You saw this technique in Chapter 6 with the `GreetingCardMaker`.

Note You can also use the ASP.NET validation controls in a Wizard without any problem. If the validation detect invalid data, they will prevent the user from clicking any of the sidebar links (to jump to another step) or they will prevent the user from continuing by clicking the Next button. However, by default the `PreviousStep` CausesValidation property set to false, which means the user *will* be allowed to step back to the previous step.

The Last Word

This chapter showed you how the rich `Calendar`, `AdRotator`, `MultiView`, and `Wizard` controls can go far beyond the limitations of ordinary HTML elements. When you're working with these controls, you don't need to think about HTML at all. Instead, you can focus on the object model that's defined by the control.

Throughout this book, you'll consider some more examples of rich controls and learn how to use them to create rich web applications that are a world apart from HTML basics. Some of the most exciting rich controls that are still ahead include the navigation controls (Chapter 13), the data controls (Chapter 16), and the security controls (Chapter 20).

Tip You might also be interested in adding third-party controls to your websites. The Internet contains many hubs for control sharing. One such location is Microsoft's own www.asp.net, which provides a control gallery where developers can submit their own ASP.NET web controls. Some of these controls are free (at least in the beta version), and others require a purchase.



User Controls and Graphics

In this chapter, you'll consider three ways to extend your web pages another notch.

First you'll tackle user controls, which give you an efficient way to reuse a block of user-interface markup—and the code that goes with it. User controls are a key tool for building modular web applications. They can also help you create consistent website designs and reuse your hard work.

Next you'll explore custom drawing with Microsoft Windows GDI+. You'll see how you can paint exactly the image you need on demand. You'll also learn the best way to incorporate a dynamic image into a web page.

Finally, you'll look at the ASP.NET Chart control, which takes your data and does the drawing for you. The end result is a polished chart that you can place in a web form, alongside your other controls and content.

User Controls

A well-built web application divides its work into discrete, independent blocks. The more modular your web application is, the easier it is to maintain your code, troubleshoot problems, and reuse key bits of functionality.

Although it's easy enough to reuse code (you simply need to pull it out of your pages and put it into separate classes), it's not as straightforward to reuse web-page markup. You can cut and paste blocks of HTML and ASP.NET control tags, but this causes endless headaches if you want to change your markup later. Instead, you need a way to wrap up web-page markup in a reusable package, just as you can wrap up ordinary C# code. The trick is to create a *user control*.

User controls look pretty much the same as ASP.NET web forms. Like web forms, they are composed of a markup portion with HTML and control tags (the .ascx file) and can optionally use a code-behind file with event-handling logic. They can also include the same range of HTML content and ASP.NET controls, and they experience the same events as the Page object (such as Load and PreRender). The only differences between user controls and web pages are as follows:

- User controls use the file extension .ascx instead of .aspx, and their code-behind files inherit from the System.Web.UI.UserControl class. In fact, the UserControl class and the Page class both inherit from the same base classes, which is why they share so many of the same methods and events, as shown in the inheritance diagram in Figure 11-1.
- The .ascx file for a user control begins with a <%@ Control %> directive instead of a <%@ Page %> directive.
- User controls can't be requested directly by a web browser. Instead, they must be embedded inside other web pages.

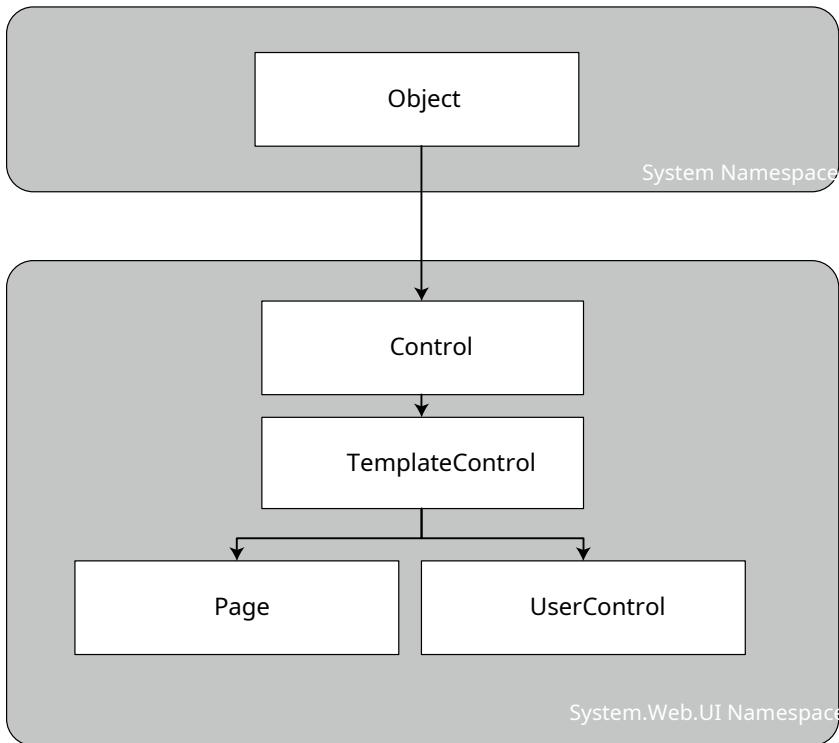


Figure 11-1. The Page and UserControl inheritance chain

Creating a Simple User Control

You can create a user control in Visual Studio in much the same way you add a web page. Just choose Website Add New Item, and select Web User Control from the list.

The following user control contains a single Label control:

```
<%@ Control Language="C#" AutoEventWireup="true"
CodeFile="Footer.ascx.cs" Inherits="Footer" %>

<asp:Label id="lblFooter" runat="server" />
```

Note that the Control directive uses the same attributes used in the Page directive for a web page, including Language, AutoEventWireup, and Inherits.

The code-behind class for this sample user control is similarly straightforward. It uses the UserControl.Load event to add some text to the label:

```
public partial class Footer : System.Web.UI.UserControl
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        lblFooter.Text = "This page was served at ";
        lblFooter.Text += DateTime.Now.ToString();
    }
}
```

To test this user control, you need to insert it into a web page. This is a two-step process. First you need to add a *Register directive* to the page that will contain the user control. You place the Register directive immediately after the Page directive. The Register directive identifies the control you want to use and associates it with a unique control prefix, as shown here:

```
<%@ Register TagPrefix="apress" TagName="Footer" Src="Footer.ascx" %>
```

The Register directive specifies a tag prefix and name. Tag prefixes group sets of related controls (for example, all ASP.NET web controls use the tag prefix *asp*). Tag prefixes are usually lowercase—technically, they are case-insensitive—and should be unique to your company or organization. The Src directive identifies the location of the user-control template file, not the code-behind file.

Second, you can now add the user control whenever you want (and as many times as you want) in the page by inserting its control tag. Consider this page example:

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeFile="FooterHost.aspx.cs" Inherits="FooterHost"%>
<%@ Register TagPrefix="apress" TagName="Footer" Src="Footer.ascx" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title> Footer Host</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <h1> A Page With a Footer</h1> <hr />
      Static Page Text <br /> <br />
      <apress:Footer id="Footer1" runat="server" />
    </div>
  </form>
</body>
</html>
```

This example (shown in Figure 11-2) demonstrates a simple way that you can create a header or footer and reuse it in all the pages in your website just by adding a user control. In the case of your simple footer, you won't save much code. However, this approach will become much more useful for a complex control with extensive formatting or several contained controls.



Figure 11-2. A page with a user-control footer

Of course, this only scratches the surface of what you can do with a user control. In the following sections, you'll learn how to enhance a control with properties, methods, and events—transforming it from a simple "include file" into a full-fledged object.

Note The Page class provides a `loadControl()` method that allows you to create a user control dynamically at runtime from an .ascx file. The user control is returned to you as a control object, which you can then add to the `Controls` collection of a container control on the web page (such as PlaceHolder or Panel) to display it on the page. This technique isn't a good substitute for declaratively using a user control, because it's more complex. However, it does have some interesting applications if you want to generate a user interface dynamically.

In Visual Studio, you have a useful shortcut for adding a user control to a page without typing the `Register` directive by hand. Start by opening the web page you want to use. Then find the .ascx file for the user control in the Solution Explorer. Drag it from the Solution Explorer and drop it onto the visual design area of your web form (not the source view area). Visual Studio will automatically add the `Register` directive for the user control, as well as an instance of the user control tag.

Another option is to configure your user control in the `web.config` file for your web application. Here's an example that registers the Footer control in this way:

```
<configuration>
<system.web>
  <pages>
    <controls>
      <add tagPrefix="apress" src="~/Footer.ascx" tagName="Footer" />
    </controls>
  </pages>
</system.web>
</configuration>
```

You can add more <add> elements to the <controls> section to register as many user controls as you want.

When you take this step, the user control is available in every page, with no Register directives required.

Visual Studio will also respect your configuration, so if you drop a user control onto a page, it will use the defined tag prefix and won't add a Register directive.

Working with Independent User Controls

Conceptually, two types of user controls exist: independent and integrated. *Independent* user controls don't interact with the rest of the code on your form. The Footer user control is one such example. Another example is a LinkMenu control that contains a list of buttons offering links to other pages. This LinkMenu user control can handle the events for all the buttons and then run the appropriate Response.Redirect() code to move to another web page. Or it can just be an ordinary HyperLink control that doesn't have any associated server-side code. Every page in the website can then include the same LinkMenu user control, enabling painless website navigation with no need to worry about frames.

Note You can use the more feature-rich navigation controls to provide website navigation. Creating your own custom controls gives you a more flexible (and more tedious) approach to providing navigation. You're not limited to using navigation controls; you can use custom controls rather than a whole site map for straightforward navigation between a few pages.

The following sample defines a simple control that presents an attractively formatted list of links. Note that the style attribute of the <div> tag (which defines fonts and formatting) has been omitted for clarity.

```
<%@ Control Language="C#" AutoEventWireup="true"
  CodeFile="LinkMenu.ascx.cs" Inherits="LinkMenu" %>
<div>
  Products:<br />
  <asp:HyperLink id="lnkBooks" runat="server"
    NavigateUrl="MenuHost.aspx?product=Books"> Books
  </asp:HyperLink> <br />
  <asp:HyperLink id="lnkToys" runat="server"
    NavigateUrl="MenuHost.aspx?product=Toys"> Toys
  </asp:HyperLink> <br />
  <asp:HyperLink id="lnkSports" runat="server"
    NavigateUrl="MenuHost.aspx?product=Sports"> Sports
  </asp:HyperLink> <br />
  <asp:HyperLink id="lnkFurniture" runat="server"
    NavigateUrl="MenuHost.aspx?product=Furniture"> Furniture
  </asp:HyperLink>
</div>
```

The links don't actually trigger any server-side code—instead, they render themselves as ordinary HTML anchor tags with a hard-coded URL.

To test this menu, you can use the following MenuHost.aspx web page. It includes two controls: the Menu control and a Label control that displays the product query string parameter. Both are positioned using a table.

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeFile="MenuHost.aspx.cs" Inherits="MenuHost"%>
<%@ Register TagPrefix="apress" TagName="LinkMenu" Src="LinkMenu.ascx" %>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title> Menu Host</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <table>
                <tr>
                    <td> <apress:LinkMenu id="Menu1" runat="server" /></td>
                    <td> <asp:Label id="lblSelection" runat="server" /></td>
                </tr>
            </table>
        </div>
    </form>
</body>
</html>
```

When the MenuHost.aspx page loads, it adds the appropriate information to the `lblSelection` control:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Request.Params["product"] != null)
    {
        lblSelection.Text = "You chose: ";
        lblSelection.Text += Request.Params["product"];
    }
}
```

Figure 11-3 shows the end result. Whenever you click a button, the page is posted back, and the text is updated.

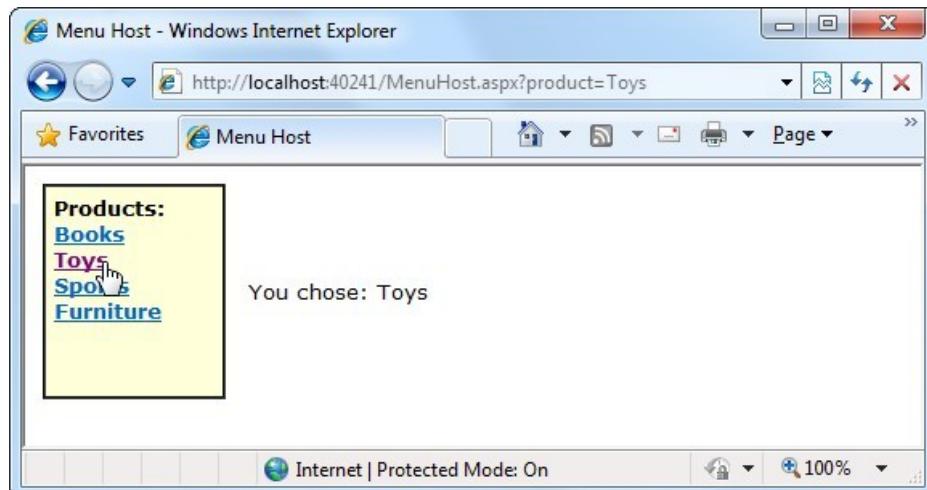


Figure 11-3. The `LinkMenu` user control

You could use the LinkMenu control to repeat the same menu on several pages. This is particularly handy when you can't use master pages to standardize layout (possibly because the pages are too different).

Working with Integrated User Controls

Integrated user controls interact in one way or another with the web page that hosts them. When you're designing these controls, the class-based design tips you learned in Chapter 4 really become useful.

A typical example is a user control that allows some level of configuration through properties. For instance, you can create a footer that supports two display formats: long date and short time. When a web page uses the footer, it will decide which display format it wants. Its code will then set the format by setting a property in the Footer class.

To make this system as clear as possible, the Footer control uses an enumeration. That way, it's clear that there are exactly two formats, and it's reasonably obvious what both formats do. It's also much more difficult to make a mistake when setting the footer.

You can create the enumeration inside the Footer class. Here's what it looks like:

```
public enum FooterFormat
{
    LongDate,
    ShortTime
}
```

Remember, an enumeration is simply a type of constant that is internally stored as an integer but is set in code by using one of the allowed names you specify. Variables that use the FooterFormat enumeration can take the value FooterFormat.LongDate or FooterFormat.ShortTime.

The next step is to add a property to the Footer class that allows the web page to retrieve or set the current format applied to the footer. The actual format is stored in a private variable called format, which is set to the long-date format by default when the control is first created. (If you're hazy on how property procedures work, feel free to review the explanation in Chapter 3.)

```
private FooterFormat format=FooterFormat.LongDate;

public FooterFormat Format
{
    get { return format; }
    set { format = value; }
}
```

Finally, the UserControl.Load event handler needs to take account of the current footer state and format the output accordingly. The following is the full Footer class code:

```
public partial class Footer : System.Web.UI.UserControl
{
    public enum FooterFormat
    { LongDate, ShortTime }

    private FooterFormat format = FooterFormat.LongDate;
    public FooterFormat Format
    {
        get { return format; }
        set { format = value; }
    }
}
```

```
protected void Page_Load(Object sender, EventArgs e)
{
    lblFooter.Text = "This page was served at ";

    if (format == FooterFormat.LongDate)
    {
        lblFooter.Text += DateTime.Now.ToString("yyyy-MM-dd");
    }
    else if (format == FooterFormat.ShortTime)
    {
        lblFooter.Text += DateTime.Now.ToString("HH:mm:ss");
    }
}
```

To test this footer, you need to create a page that modifies the Format property of the Footer user control. Figure 11-4 shows an example page, which automatically sets the Format property for the user control to match a radio button selection whenever the page is posted back.



Figure 11-4. The modified footer

Note that the user-control property is modified in the Page.Load event handler, not the cmdRefresh.Click event handler. The reason is that the Load event occurs before the user control has been rendered each time the page is created. The Click event occurs after the user control has been rendered, and though the property change is visible in your code, it doesn't affect the user control's HTML output, which has already been added to the page.

```

public partial class FooterHost : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        if (optLong.Checked)
        {
            Footer1.Format = Footer.FooterFormat.LongDate;
        }
        else if (optShort.Checked)
        {
            Footer1.Format = Footer.FooterFormat.ShortTime;
        }
        else
        {
            // The default value in the Footer class will apply.
        }
    }
}

```

You can also set the initial appearance of the footer in the control tag:

```
<apress:Footer Format="ShortTime" id="Footer1" runat="server" />
```

As with all web-control markup, ASP.NET converts strings (such as "ShortTime") to the corresponding enumeration value without a problem.

Using User-Control Events

Another way that communication can occur between a user control and a web page is through events. With methods and properties, the user control reacts to a change made by the web-page code. With events, the story is reversed: the user control notifies the web page about an action, and the web-page code responds.

Creating a web control that uses events is fairly easy. In the following example, you'll see a version of the LinkMenu control that uses events. Instead of navigating directly to the appropriate page when the user clicks a button, the control raises an event, which the web page can choose to handle.

The first step to create this control is to define the events. Remember, to define an event, you must first choose an event signature. The .NET standard for events specifies that every event should use two parameters. The first one provides a reference to the control that sent the event, while the second incorporates any additional information. This additional information is wrapped into a custom EventArgs object, which inherits from the System.EventArgs class. (If your event doesn't require any additional information, you can just use the predefined EventArgs class, which doesn't contain any additional data. Many events in ASP.NET, such as Page.Load or Button.Click, follow this pattern.) You can refer to Chapter 4 for a quick overview of how to use events in .NET.

The LinkMenu2 control uses a single event, which indicates when a link is clicked:

```

public partial class LinkMenu2 : System.Web.UI.UserControl
{
    public event EventHandler LinkClicked;

    ...
}

```

This code defines an event named LinkClicked. The LinkClicked event has the signature specified by the System.EventHandler delegate, which includes two parameters—the event sender and an ordinary EventArgs

object. That means if you wanted to create an event handler for the LinkClicked event (say, in a web page that features the LinkMenu2 control), your event handler must look like this:

```
protected void LinkMenu_LinkClicked(object sender, EventArgs e)
{ ... }
```

This takes care of defining the event, but what about raising it? This part is easy. To fire the event, the LinkMenu2 control simply calls the event by name and passes in the two parameters, like this:

```
// Raise the LinkClicked event, passing a reference to
// the current object (the sender) and an empty EventArgs object.
LinkClicked(this, EventArgs.Empty);
```

The LinkMenu2 control actually needs a few more changes. The original version used the HyperLink control. This won't do, because the HyperLink control doesn't fire an event when the link is clicked. Instead, you'll need to use the LinkButton. The LinkButton fires the Click event, which the LinkMenu2 control can intercept, and then raises the LinkClicked event to the web page.

The following is the full user-control code:

```
public partial class LinkMenu2 : System.Web.UI.UserControl
{
    public event EventHandler LinkClicked;

    protected void Ink_Click(object sender, EventArgs e)
    {
        // One of the LinkButton controls has been clicked.
        // Raise an event to the page.
        if (LinkClicked != null)
        {
            LinkClicked(this, EventArgs.Empty);
        }
    }
}
```

Notice that before raising the LinkClicked event, the LinkMenu2 control needs to test the LinkClicked event for a null reference. A null reference exists if no event handlers are attached to the event. In this case, you shouldn't try to raise the event, because it would only cause an error.

You can create a page that uses the LinkMenu2 control and add an event handler. Unfortunately, you won't be able to connect these event handlers by using the Visual Studio Properties window, because the Properties window won't show the custom events that the user control provides. Instead, you'll need to modify the LinkMenu2 tag directly, as shown here:

```
<apress:LinkMenu2 id="links" runat="server" OnLinkClicked="LinkClicked" />
```

And here's the event handler that responds in the web page:

```
protected void LinkClicked(object sender, EventArgs e)
{
    lblClick.Text = "Click detected.";
}
```

Note You won't be able to create user-control event handlers through the Visual studio Properties window instead, you'll need to type in your event handler by hand.

Figure 11-5 shows the result.

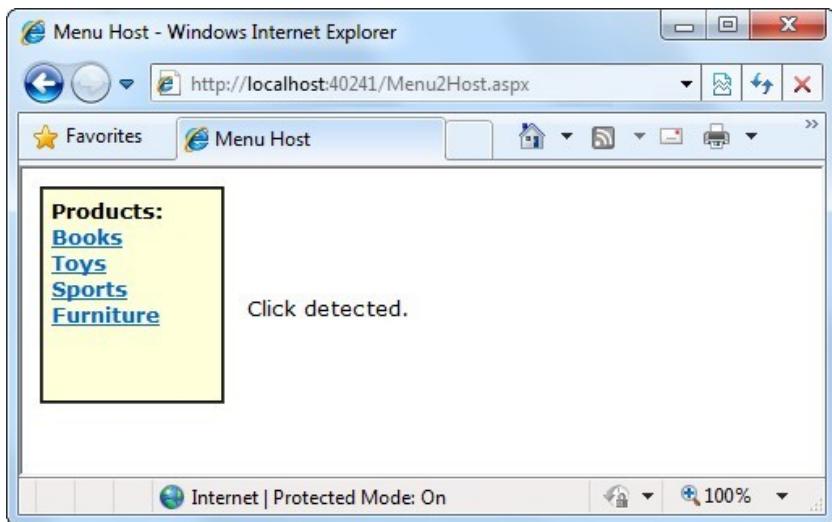


Figure 11-5. Using the LinkMenu2 user control

Conceptually, this approach should give your web page more power to customize how the user control works. Unfortunately, that's not the case at the moment, because a key piece of information is missing. When the LinkClicked event occurs, the web page has no way of knowing what link was clicked, which prevents it from taking any kind of reasonable action. The only way to solve this problem is to create a more intelligent event that can transmit some information through event arguments. You'll see how in the next section.

Passing Information with Events

In the current LinkMenu2 example, no custom information is passed along with the event. In many cases, however, you want to convey additional information that relates to the event. To do so, you need to create a custom class that derives from EventArgs.

The LinkClickedEventArgs class that follows allows the LinkMenu2 user control to pass the URL that the user selected through a Url property. It also provides a Cancel property. If set to true, the user control will stop its processing immediately. But if Cancel remains false (the default), the user control will send the user to the new page. This way, the user control still handles the task of redirecting the user, but it allows the web page to plug into this process and change it or stop it (for example, if there's unfinished work left on the current page).

```
public class LinkClickedEventArgs : EventArgs
{
    public string Url {get; set;}
    public bool Cancel {get; set;}

    public LinkClickedEventArgs(string url)
    {
        Url = url;
    }
}
```

To use this custom EventArgs class, you need to modify the definition of the LinkClicked event so it uses the LinkClickedEventArgs object. Here's the change you need to make inside the LinkMenu2 class:

```
public event EventHandler <LinkClickedEventArgs> LinkClicked;
```

It's the magic of generics that makes this work. Essentially, you're configuring the generic EventHandler class to use the EventArgs class you want—in this case, LinkClickedEventArgs.

Next, your user control code for raising the event needs to submit the required information when calling the event. But how does the user control determine what link was clicked? The trick is to switch from the LinkButton.Click event to the LinkButton.Command event. The Command event automatically gets the CommandArgument that's defined in the tag. So if a web page defines your LinkButton controls like this:

```
<asp:LinkButton ID="InkBooks" runat="server"
    CommandArgument="Menu2Host.aspx?product=Books" OnCommand="Ink_Command"> Books
</asp:LinkButton> <br />
<asp:LinkButton ID="InkToys" runat="server"
    CommandArgument="Menu2Host.aspx?product=Toys" OnCommand="Ink_Command"> Toys
</asp:LinkButton> <br />
<asp:LinkButton ID="InkSports" runat="server"
    CommandArgument="Menu2Host.aspx?product=Sports" OnCommand="Ink_Command"> Sports
</asp:LinkButton> <br />
<asp:LinkButton ID="InkFurniture" runat="server"
    CommandArgument="Menu2Host.aspx?product=Furniture" OnCommand="Ink_Command">
Furniture</asp:LinkButton>
```

the LinkMenu2 control can pass the link along to the web page like this, when it raises the event:

```
LinkClickedEventArgs args = new LinkClickedEventArgs((string)e.CommandArgument);
LinkClicked(this, args);
```

Here's the complete user-control code. It implements one more feature. After the event has been raised and handled by the web page, the LinkMenu2 checks the Cancel property. If it's false, the code goes ahead and performs the redirect by using Response.Redirect().

```
public partial class LinkMenu2 : System.Web.UI.UserControl
{
    public event EventHandler <LinkClickedEventArgs> LinkClicked;

    protected void Ink_Command(object sender, CommandEventArgs e)
    {
        // One of the LinkButton controls has been clicked.
        // Raise an event to the page.
        if (LinkClicked != null)
        {
            // Pass along the link information.
            LinkClickedEventArgs args =
                new LinkClickedEventArgs((string)e.CommandArgument);
            LinkClicked(this, args);

            // Perform the redirect.
            if (!args.Cancel)
            {
                // Notice we use the Url from the LinkClickedEventArgs
                // object, not the original link. That means the web page
                // can change the link if desired before the redirect.
            }
        }
    }
}
```

```
        Response.Redirect(args.Url);  
    }  
}
```

Finally, you need to update the code in the web page (where the user control is placed) so that its event handler uses the new signature. In the following code, the LinkClicked event handler checks the URL and allows it in all cases except one:

```
protected void LinkClicked(object sender, LinkClickedEventArgs e)
{
    if (e.Url == "Menu2Host.aspx?product=Furniture")
    {
        lblClick.Text = "This link is not allowed.";
        e.Cancel = true;
    }
    else
    {
        // Allow the redirect, and don't make any changes to the URL.
    }
}
```

If you click the Furniture link, you'll see the message shown in Figure 11-6. This confirms that your user control can communicate with the page—not only does it raise an event, but it also transmits some useful information.



Figure 11-6. Handling a user-control event in the page

Dynamic Graphics

One of the features of the .NET Framework is Windows GDI+, a set of classes designed for drawing images. You can use GDI+ in a Windows or an ASP.NET application to create dynamic graphics. In a Windows application, the graphics you draw would be copied to a window for display. In ASP.NET, your code can render the graphics you want and send them directly to the client browser.

In general, using GDI+ code to draw a graphic is slower than using a ready-made image file. However, GDI+ gives you much more freedom. For example, you can tailor your image to suit a particular purpose, incorporating information such as the date or current username. You can also mingle text, shapes, and other bitmaps to create a complete picture.

the htML5 CaNVaS

if you've played with HTML5, you know that the new language adds a feature called the Canvas. Quite simply, the Canvas is a client-side drawing tool. it's a rectangular drawing surface that you define on the page, and then fill by using Javascript commands.

Because the Canvas uses Javascript, it performs its drawing in the client's browser (whereas GDI+ does work on the web server). This has important consequences—it means, for example, that the Canvas is certainly slower than GDI+, and that the Canvas won't be supported on all browsers and devices (unless which converts its work to an ordinary graphic that all browsers can understand). However, the Canvas has one significant advantage: because the work happens on the client, developers can create dynamic graphics that respond to user actions. For example, it's easy to make a drawing application with the Canvas, where Javascript to draw a line wherever the user scribbles. The same feat would be impossible with GDI+, because you would need to post the page back to the web server to update the image after every mouse move.

The bottom line is this: GDI+ is the best solution for rendering a static graphic, while the Canvas is a promising tool for creating Javascript-powered applets that require dynamic drawing capabilities. if you want to learn more about the Canvas and HTML5, check out *HTML5: The Missing Manual* (o'Reilly, 2011).

Creating a Basic Drawing

Before you get started, you should import the System.Drawing namespace, which has the classes that GDI+ uses: using System.Drawing;

You need to follow four basic steps when using GDI+. First you have to create an in-memory bitmap. This is the drawing space where you'll create your masterpiece. To create the bitmap, declare a new instance of the System.Drawing.Bitmap class. You must specify the height and width of the image in pixels. Be careful—don't make the bitmap larger than required, or you'll needlessly waste memory.

```
// Create an in-memory bitmap where you will draw the image.  
// The Bitmap is 300 pixels wide and 50 pixels high.  
Bitmap image = new Bitmap(300, 50);
```

The next step is to create a GDI+ graphics context for the image, which is represented by a System.Drawing.Graphics object. This object provides the methods that allow you to render content to the in-memory bitmap. To create a Graphics object from an existing Bitmap object, you just use the static Graphics.FromImage() method, as shown here:

```
Graphics g = Graphics.FromImage(image);
```

Note The Graphics.FromImage() method works with any image object. Classes such as Bitmap derive from Image, so they work fine.

Now comes the interesting part. Using the methods of the `Graphics` class, you can draw text, shapes, and images on the bitmap. Table 11-1 lists some of the most fundamental methods of the `Graphics` class. The methods that begin with the word *Draw* draw outlines, while the methods that begin with the word *Fill* draw solid regions. The only exceptions are the `DrawString()` method, which draws filled-in text using a font you specify, and the methods for copying bitmap images, such as `DrawIcon()` and `DrawImage()`.

Table 11-1. Drawing Methods of the `Graphics` Class

Method	Description
<code>DrawArc()</code>	Draws an arc representing a portion of an ellipse specified by a pair of coordinates, a width, and a height (or some other combination of information, if you use one of the overloaded versions of this method).
<code>DrawBezier()</code> and <code>DrawBeziers()</code>	Draws the infamous and attractive Bezier curve, which is defined by four control points.
<code>DrawClosedCurve()</code>	Draws a curve and then closes it off by connecting the end points.
<code>DrawCurve()</code>	Draws a curve (technically, a cardinal spline).
<code>DrawEllipse()</code>	Draws an ellipse defined by a bounding rectangle specified by a pair of coordinates, a height, and a width.
<code>DrawIcon()</code> and <code>DrawIconUnstretched()</code>	Draws the icon represented by an <code>Icon</code> object and (optionally) stretches it to fit a given rectangle.
<code>DrawImage()</code> and <code>DrawImageUnscaled()</code>	Draws the image represented by an <code>Image</code> -derived object (such as a <code>Bitmap</code> object) and (optionally) stretches it to fit a given rectangle.
<code>DrawLine()</code> and <code>DrawLines()</code>	Draws one or more lines. Each line connects the two points specified by a coordinate pair.
<code>DrawPie()</code>	Draws a “piece of pie” shape defined by an ellipse specified by a coordinate pair, a width, a height, and two radial lines.
<code>DrawPolygon()</code>	Draws a multisided polygon defined by an array of points.
<code>DrawRectangle()</code> and <code>DrawRectangles()</code>	Draws one or more ordinary rectangles. Each rectangle is defined by a starting coordinate pair, a width, and a height.
<code>DrawString()</code>	Draws a string of text in a given font.
<code>DrawPath()</code>	Draws a more complex shape that's defined by the <code>Path</code> object.
<code>FillClosedCurve()</code>	Draws a curve, closes it off by connecting the end points, and fills it.
<code>FillEllipse()</code>	Fills the interior of an ellipse.
<code>FillPie()</code>	Fills the interior of a “piece of pie” shape.
<code>FillPolygon()</code>	Fills the interior of a polygon.
<code>FillRectangle()</code> and <code>FillRectangles()</code>	Fills the interior of one or more rectangles.
<code>FillPath()</code>	Fills the interior of a complex shape that's defined by the <code>Path</code> object.

When calling the `Graphics` class methods, you need to specify several parameters to indicate the pixel coordinates for what you want to draw. For example, when drawing a rectangle, you need to specify the location of the top-left corner and its width and height. Here's an example of how you might draw a solid rectangle in yellow:

```
// Draw a rectangle starting at location (0, 0)
// that is 300 pixels wide and 50 pixels high.
g.FillRectangle(Brushes.Yellow, 0, 0, 300, 50);
```

When measuring pixels, the point `(0, 0)` is the top-left corner of your image in `(x, y)` coordinates. The `x` coordinate increases as you go farther to the right, and the `y` coordinate increases as you go farther down. In the current example, the image is 300 pixels wide and 50 pixels high, which means the point `(299, 49)` is the bottom-right corner.

Note This code performs its drawing on the in-memory `Bitmap` object created earlier. Until this image is rendered (a skill you'll pick up shortly), you won't actually see anything on the web page.

You'll also notice that you need to specify either a `Brush` or a `Pen` object when you draw most content. (Both of these classes are defined in the `System.Drawing` namespace, alongside the `Graphics` class.) Methods that draw shape outlines require a `Pen`, while methods that draw filled-in shapes require a `Brush`. You can create your own custom `Pen` and `Brush` objects, but .NET provides an easier solution with the `Brushes` and `Pens` classes. These classes expose static properties that provide various `Brushes` and `Pens` for different colors. For example, `Brushes.Yellow` returns a `Brush` object that fills shapes with a solid yellow color, and `Pens.Yellow` returns a `Pen` object that draws shape outlines in the same solid yellow color.

When the image is complete, you can send it to the browser by using the `Image.Save()` method. Conceptually, you "save" the image to the browser's response stream. It then gets sent to the client and displayed in the browser.

```
// Render the image to the HTML output stream.
Response.ContentType = "image/gif";
image.Save(Response.OutputStream,
    System.Drawing.Imaging.ImageFormat.Gif);
```

Tip You can save an image to any valid stream, including the `Filestream` class described in Chapter 17. This technique allows you to save dynamically generated images to disk so you can use them later in other web pages.

Finally, you should explicitly release your image and graphics context when you're finished, because both of these objects hold onto some unmanaged resources that might not be released right away if you don't:

```
g.Dispose();
image.Dispose();
```

Using GDI+ is a specialized technique, and its more advanced features are beyond the scope of this book. However, you can learn a lot by considering a couple of straightforward examples.

Drawing a Custom Image

Using the techniques you've learned, it's easy to create a simple web page that uses GDI+. The next example uses GDI+ to render some text in a bordered rectangle with a happy-face graphic next to it.

Here's the code you'll need:

```
protected void Page_Load(Object sender, EventArgs e)
{
    // Create an in-memory bitmap where you will draw the image.
    // The Bitmap is 300 pixels wide and 50 pixels high.
    Bitmap image = new Bitmap(300, 50);

    // Get the graphics context for the bitmap.
    Graphics g = Graphics.FromImage(image);

    // Draw a solid yellow rectangle with a red border.
    g.FillRectangle(Brushes.LightYellow, 0, 0, 300, 50);
    g.DrawRectangle(Pens.Red, 0, 0, 299, 49);

    // Draw some text using a fancy font.
    Font font = new Font("Alba Super", 20, FontStyle.Regular);
    g.DrawString("This is a test.", font, Brushes.Blue, 10, 0);

    // Copy a smaller gif into the image from a file.
    // This code assumes smiley.gif is in the root website root.
    System.Drawing.Image icon =
        System.Drawing.Image.FromFile(Server.MapPath("smiley.gif"));
    g.DrawImageUnscaled(icon, 240, 0);

    // Render the entire bitmap to the HTML output stream.
    Response.ContentType = "image/gif";
    image.Save(Response.OutputStream,
        System.Drawing.Imaging.ImageFormat.Gif);

    // Clean up.
    g.Dispose();
    image.Dispose();
}
```

This code is easy to understand. It follows the basic pattern set out earlier—it creates the in-memory Bitmap, gets the corresponding Graphics object, performs the painting, and then saves the image to the response stream. This example uses the FillRectangle(), DrawRectangle(), DrawString(), and DrawImageUnscaled() methods to create the complete drawing shown in Figure 11-7.

Tip Because this image is generated on the web server, you can use any font that is installed on the server. The client doesn't need to have the same font, because the client receives the text as a rendered image.



Figure 11-7. Drawing a custom image

Placing Custom Images Inside Web Pages

The `Image.Save()` approach demonstrated so far has one problem. When you save an image to the response stream, you overwrite whatever information ASP.NET would otherwise use. If you have a web page that includes other content and controls, this content won't appear at all in the final web page. Instead, the dynamically rendered graphics replace it.

Fortunately, this has a simple solution: you can link to a dynamically generated image by using the HTML `` tag or the `Image` web control. But instead of linking your image to a fixed image file, link it to the `.aspx` file that generates the picture.

For example, you could create a file named `GraphicalText.aspx` that writes a dynamically generated image to the response stream. In another page, you could show the dynamic image by adding an `Image` web control and setting the `ImageUrl` property to `GraphicalText.aspx`. In fact, you'll even see the image appear in Visual Studio's design-time environment before you run the web page!

When you use this technique to embed dynamic graphics in web pages, you also need to think about how the web page can send information to the dynamic graphic. For example, what if you don't want to show a fixed piece of text, but instead you want to generate a dynamic label that incorporates the name of the current user? (In fact, if you do want to show a fixed piece of text, it's probably better to create the graphic ahead of time and store it in a file, rather than generating it by using GDI+ code each time the user requests the page.) One solution is to pass the information by using the query string, as described in Chapter 8. The page that renders the graphic can then check for the query string information it needs.

Here's how you'd rewrite the dynamic graphic generator with this in mind:

```
// Get the username.
if (Request.QueryString["Name"] == null)
{
    // No name was supplied.
    // Don't display anything.
}
else
{
    string name = Request.QueryString["Name"];
}
```

```

// Create an in-memory bitmap where you will draw the image.
Bitmap image = new Bitmap(300, 50);

// Get the graphics context for the bitmap.
Graphics g = Graphics.FromImage(image);

g.FillRectangle(Brushes.LightYellow, 0, 0, 300, 50);
g.DrawRectangle(Pens.Red, 0, 0, 299, 49);

// Draw some text based on the query string.
Font font = new Font("Alba Super", 20, FontStyle.Regular);
g.DrawString(name, font, Brushes.Blue, 10, 0);

// Render the entire bitmap to the HTML output stream.
Response.ContentType = "image/gif";
image.Save(Response.OutputStream,
    System.Drawing.Imaging.ImageFormat.Gif);

g.Dispose();
image.Dispose();
}

```

Conceptually, this code isn't much different from the examples you've seen before. The only change is that one piece of information—the string that's used with the `DrawString()` method—is retrieved from the query string.

Figure 11-8 shows a page that uses this dynamic graphic page, along with two Label controls. The page passes the query string argument Joe Brown to the page. The full `ImageUrl` thus becomes `GraphicalText.aspx?Name=Joe%20Brown`, as shown here:

```

<asp:Label id="Label1" runat="server"> Here is some content.</asp:Label>
<br /> <br />
<asp:Image id="Image1" runat="server"
    ImageUrl="GraphicalText.aspx?Name=Joe%20Brown"> </asp:Image>
<br /> <br />
<asp:Label id="Label2" runat="server"> Here is some more content.</asp:Label>

```



Figure 11-8. Mingling custom images and controls on the same page

It's possible that you might need to send more information or more-complex information to the page that draws the image. For example, you might want to pass a data object to a page that draws a pie chart. In this case, the query string isn't good enough, and you'll need to use a different type of state management. One option is session state, as described in Chapter 8. (Another option is to avoid the problem altogether with ASP.NET's own Chart control, as you'll see later in this chapter.)

Choosing Image Format and Quality

When you render an image, you can also choose the format you want to use. JPEG offers the best color support and graphics, although it uses compression that can lose detail and make text look fuzzy. GIF (the standard used in the examples so far) is often a better choice for graphics containing text, but it doesn't offer good support for color. In .NET, every GIF uses a fixed palette with 256 generic colors. If you use a color that doesn't map to one of these presets, the color will be dithered, leading to a less-than-optimal graphic.

However, the best format choice is PNG. PNG is an all-purpose format that always provides high quality by combining the lossless compression of GIFs with the rich color support of JPEGs. (The only hiccup is that versions of Internet Explorer before version 7 don't correctly deal with PNG content returned from a web page. If you're using one of these old versions of IE, you won't see the picture content—instead, you'll receive a message prompting you to download the picture and open it in another program. To sidestep this problem, you need to use the `` tag approach shown in the previous example.)

You also need to be aware of one more quirk when using PNG—you can't use the `Bitmap.Save()` method shown in earlier examples. If you do, an error will occur. Technically, the problem is that the `Save()` method requires a *seekable* stream—a stream where the position can be changed at will. That's because .NET needs to be able to move back and forth through the picture content while it's being generated.

The solution is easy to implement, if a little awkward. Instead of saving directly to `Response.OutputStream`, you can create a `System.IO.MemoryStream` object, which represents an in-memory buffer of data. The `MemoryStream` is always seekable, so you can save the image to this object. After you've performed this step, you can easily copy the data from the `MemoryStream` to the `Response.OutputStream`. The only disadvantage is that this technique requires more memory because the complete rendered content of the graphic needs to be held in memory at once. However, the graphics you use in web pages generally aren't that large, so you probably won't observe any reduction in performance.

To implement this solution, start by importing the `System.IO` namespace:

```
using System.IO;
```

Now you can replace the previous example with this modified code that saves the image in PNG format. The changed lines are highlighted.

```
// Get the username.
if (Request.QueryString["Name"] == null)
{
    // No name was supplied.
    // Don't display anything.
}
else
{
    string name = Request.QueryString["Name"];

    // Create an in-memory bitmap where you will draw the image.
    Bitmap image = new Bitmap(300, 50);

    // Get the graphics context for the bitmap.
    Graphics g = Graphics.FromImage(image);
```

```

g.FillRectangle(Brushes.LightYellow, 0, 0, 300, 50);
g.DrawRectangle(Pens.Red, 0, 0, 299, 49);

// Draw some text based on the query string.
Font font = new Font("Alba Super", 20, FontStyle.Regular);
g.DrawString(name, font, Brushes.Blue, 10, 0);

Response.ContentType = "image/png";

// Create the PNG in memory.
MemoryStream mem = new MemoryStream();
image.Save(mem, System.Drawing.Imaging.ImageFormat.Png);

// Write the MemoryStream data to the output stream.
mem.WriteTo(Response.OutputStream);

g.Dispose();
image.Dispose();
}

```

Note You'll learn more about streams when you tackle file access in Chapter 17.

Quality isn't determined by only the image format. It also depends on the way you draw the image content onto the in-memory bitmap. GDI+ allows you to choose between optimizing your drawing code for appearance or speed. When you choose to optimize for the best appearance, .NET uses extra rendering techniques such as antialiasing to improve the drawing.

Antialiasing smooths jagged edges in shapes and text. It works by adding shading at the border of an edge. For example, gray shading might be added to the edge of a black curve to make a corner look smoother. Technically, antialiasing blends a curve with its background. Figure 11-9 shows a close-up of an antialiased ellipse.



Figure 11-9. Antialiasing with an ellipse

To use smoothing in your applications, you set the `SmoothingMode` property of the `Graphics` object. You can choose between `None`, `HighSpeed` (the default), `AntiAlias`, and `HighQuality` (which is similar to `AntiAlias` but uses other, slower optimizations that improve the display on LCD screens). The `Graphics.SmoothingMode` property is one of the few stateful `Graphics` class members. This means you set it before you begin drawing, and it applies to any text or shapes you draw in the rest of the paint session (until the `Graphics` object is released).

```
g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
```

Tip Antialiasing makes the most difference when you're displaying curves. That means it will dramatically improve the appearance of ellipses, circles, and arcs, but it won't make any difference with straight lines and rectangles.

You can also use antialiasing with fonts to soften jagged edges on text. You can set the `Graphics.TextRenderingHint` property to ensure optimized text. You can choose between `SingleBitPerPixelGridFit` (fastest performance and lowest quality), `AntiAliasGridFit` (better quality but slower performance), and `ClearTypeGridFit` (the best quality on an LCD display). Or you can use the `SystemDefault` value to apply whatever font-smoothing settings the user has configured. `SystemDefault` is the default setting, and the default system settings for most computers enable text antialiasing. Even if you don't set this, your dynamically rendered text will probably be drawn in high quality. However, because you can't necessarily control the system settings of the web server, it's a good practice to specify this setting explicitly if you need to draw text in an image.

The Chart Control

As you saw in the previous section, the GDI+ drawing framework lets you create any sort of graphical content you can imagine. Crafty developers use it to build made-to-measure graphs, charts, and infographics. (In fact, you can find one such example in *Pro ASP.NET 4.5 in C#*.) However, although such examples are conceptually straightforward—your code takes some numbers, makes some calculations, and draws bars or pie slices to represent the results—getting a nice, visually appealing result is tedious. Creating a good chart takes lots of fiddling and plenty of testing.

Fortunately, there's a huge shortcut. Thanks to ASP.NET's Chart control, developers rarely need to build their own charts from scratch. Behind the scenes, the Chart control uses GDI+ functions to create an image that you then display in your web page. The difference is that you don't need to touch any of the actual drawing code. Instead, you simply supply the chart with the right data, choose the type of chart you want, set some other properties to customize the way that it's rendered, and enjoy the final result.

Note The Chart is packed with a staggering array of features, including some impressive customizability and performance-enhancement options. In fact, it would take several full chapters to cover the Chart's features. In the following section, you'll simply get a taste of how the Chart control works and what it can do. You'll revisit Chapter 16 (where you'll use it with the data extracted from a database), but if you want to learn more, you'll need to start experimenting or read the Chart documentation on the MSDN website at <http://msdn.microsoft.com/library/dd489242>.

Creating a Basic Chart

When you add a brand new Chart control to a page, Visual Studio registers the Chart control, so it's ready to use in your web page. It does that by adding a directive like this at the beginning of your web page markup:

```
<%@ Register Assembly="System.Web.DataVisualization, Version=4.0.0.0, ..." TagPrefix="asp" %>
```

Visual Studio also adds a new instance of the Chart control to your page, using markup like this:

```
<asp:Chart ID="Chart1" runat="server">
<Series>
  <asp:Series Name="Series1">
  </asp:Series>
```

```
</Series>
<ChartAreas>
  <asp:ChartArea Name="ChartArea1">
    </asp:ChartArea>
  </ChartAreas>
</asp:Chart>
```

Keen eyes will notice that this chart is split into two key sections. The first section, `<Series>`, holds the data that you want to plot on the chart and configures how it will be presented. It's here where you set everything from the type of chart (for example, line graph versus pie chart) to finer details such as the legend text and the style of markers you use for data points. The second section, `<ChartAreas>`, lets you influence the rest of your chart—in other words, the area that appears behind and around your data. You can use this section to control chart shading, apply a 3D effect, and set the scale of the axes.

With this in mind, it doesn't take much work to create a super-simple chart. The following example creates a pie chart with four slices:

```
<h3>.NET Fans - Favorite Pastime Survey</h3>
<asp:Chart ID="passtimeChart" runat="server">
  <Series>
    <asp:Series ChartType="Pie">
      <Points>
        <asp:DataPoint AxisLabel="Programming the Web" YValues="42" />
        <asp:DataPoint AxisLabel="Building Robots" YValues="18" />
        <asp:DataPoint AxisLabel="Playing Video Games" YValues="17" />
        <asp:DataPoint AxisLabel="Playing Foosball" YValues="5" />
      </Points>
    </asp:Series>
  </Series>
  <ChartAreas>
    <asp:ChartArea Name="chartArea">
    </asp:ChartArea>
  </ChartAreas>
</asp:Chart>
```

To build this chart, you begin by creating one series of data. When you define the series, you choose the type of chart by setting the `ChartType` property to one of 35 different chart types:

```
<asp:Series ChartType="Pie">
  ...
</asp:Series>
```

Note More-complex charts have multiple series. For example, if you create a line chart comparing the sales numbers over time in different stores, each store would get its own line, and each line would be a series.

Inside the series, you create a collection of data points. To create a four-slice pie chart requires four data points. But you can switch the display to a completely different (but equally effective) column graph by changing one series property:

```
<asp:Series ChartType="Column">
```

Figure 11-10 compares the pie chart and column chart.

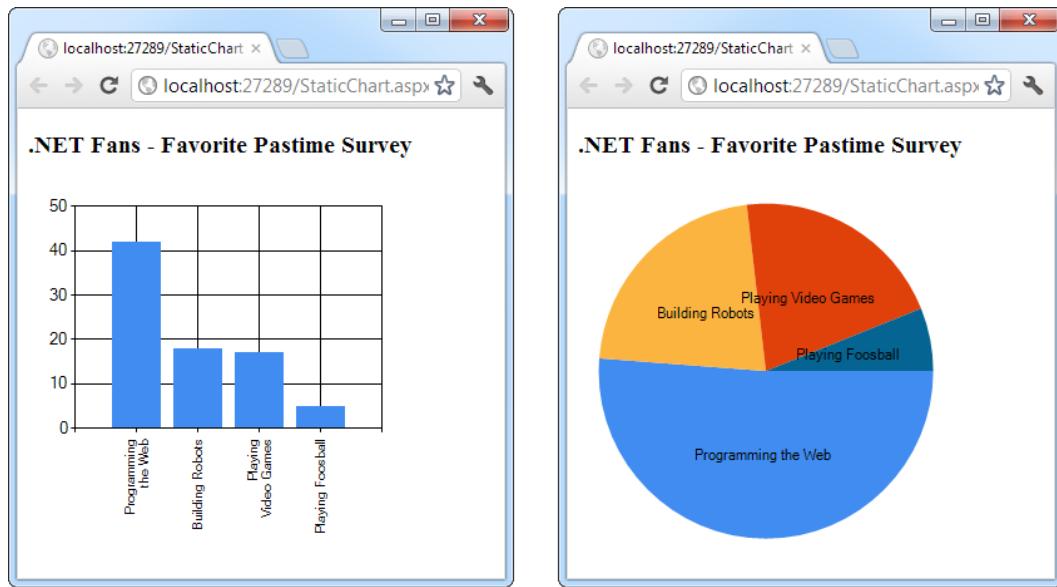


Figure 11-10. The same data in two chart types

When you run this example and look at the generated HTML markup for the page, you'll find that the chart consists of a wordy `` tag that looks something like this:

```

```

Here, ASP.NET is using a variation of the technique you saw in the custom GDI+ examples. It adds an `` tag that points to the ASP.NET page that generates the image content you want. The difference in this case is that ASP.NET uses a virtual page, rather than a web form that's actually stored on your website. (In other words, there is no actual `ChartImg.axd` file. When ASP.NET receives a request for this page, it knows that it's being asked to perform charting, and it triggers the corresponding chart-drawing code.)

Using Other Ways to Get Chart Data

Obviously, a realistic application is unlikely to record its data in the page markup. Instead, it's much more likely that you will generate the data with some calculations in code or simply fetch it out of a database. As long as you give your series a name, you can access it in code and add all the data points you need. For example, if your series looks like this:

```
<asp:Series Name="passtimes" ChartType="Pie" />
```

You can programmatically create the same chart you saw earlier with this code:

```
protected void Page_Load(Object sender, EventArgs e)
{
    // Find the right series.
    var passtimeSeries = passtimeChart.Series["passtimes"];
```

```

// Add the points to the series.
passtimeSeries.Points.AddXY("Programming the Web", 42);
passtimeSeries.Points.AddXY("Building Robots ", 18);
passtimeSeries.Points.AddXY("Playing Video Games", 17);
passtimeSeries.Points.AddXY("Playing Foosball", 5);
}

```

With this technique in place, all you need to do is rework the code so that it's retrieving or calculating the data you really want to use. As long as you size it to a large enough size, the ASP.NET Chart control can handle hundreds or thousands of data points.

Often the data you want to chart will come from an external source, such as a file or a database. Later in this book, you'll learn how to read data from both types of storage. And after you know how to read your data, a simple loop is often all you need to copy it into a chart, using the preceding series-accessing technique. However, if you're dealing with a database, you have the option of using an even more powerful shortcut, called data-binding, to copy your all data straight into a chart without writing more than a line or two of code. You'll explore data binding in detail starting in Chapter 15.

Tip The real chart magic appears when you combine your business data with the snazzy formatting that makes it stand out. And although we don't cover it in this chapter, the Chart control is outfitted with a dizzying array of formatting features to customize every last graphical detail. One good way to get an idea of its formatting power is to download Microsoft's Chart control samples, which include 200 examples of charts in action. You can find the samples at <http://tinyurl.com/chartsamples>. Incidentally, the samples are provided in both ASP.NET and Windows Forms versions, because there is a Windows version of the Chart control for desktop applications that is virtually identical to the ASP.NET version you'll use.

The Last Word

In this chapter, you put three more tools in your ASP.NET toolkit. First you saw how user controls allow you to reuse a block of user interface in more than one web page. Next you considered how custom drawing allows you to create made-to-measure graphics. Finally, you saw how you could get professionally tailored charts with very little effort by taking advantage of the Chart control.

In the following chapter, you'll learn about themes and master pages—two features that complement user controls and give you even more ways to standardize the look and feel of your web pages. Themes are more fine-grained than user controls—they group together formatting presets that you can apply to individual controls to ensure a slick, consistent style throughout your application. Master pages are broader than user controls—they allow you to define a standardized page template that you can apply to lock down the appearance and layout of multiple pages, giving you complete consistency. Learning how to mix all these ingredients is part of the fine art of ASP.NET programming.



Website Navigation

You've already learned several simple ways to send a website visitor from one page to another. The most straightforward approach is to add ordinary HTML links, using either the `<a>` element or the `HyperLink` web control. These links let users click their way through the pages of your site. Or, if you want to trigger a page change in response to some other action, your code can call the handy `Response.Redirect()` method or the `Server.Transfer()` method at any time. Both methods are detailed in Chapter 5.

All these techniques are fundamental parts of ASP.NET web design. But as your website becomes more sophisticated, it gets more-complex navigational requirements—ones that can't be satisfied with a pile of ordinary links. Instead, professional applications need a complete navigation *system* that allows users to surf through a hierarchy of pages, without forcing you to copy a large block of markup or write the same tedious navigation code in every page.

Fortunately, ASP.NET includes a navigation model that makes it easy to let users surf through your web applications. Before you can use this model, you need to determine the hierarchy of your website—in other words, how pages are logically organized into groups. You then *define* that structure in a dedicated file and *bind* that information to specialized navigation controls. Best of all, these navigation controls include nifty widgets such as the `TreeView` and `Menu`.

In this chapter, you'll learn everything you need to know about the site map model and the navigation controls that work with it.

Site Maps

If your website has more than a handful of pages, you'll probably want some sort of navigation system to let users move from one page to the next. Obviously, you can use the ASP.NET toolkit of controls to implement almost any navigation system, but this requires that *you* perform all the hard work. Fortunately, ASP.NET has a set of navigation features that can simplify the task dramatically.

As with all the best ASP.NET features, ASP.NET navigation is flexible, configurable, and pluggable. It consists of three components:

- A way to define the navigation structure of your website. This part is the XML site map, which is (by default) stored in a file.
- A convenient way to read the information in the site map file and convert it to an object model. The `SiteMapDataSource` control and the `XmlSiteMapProvider` perform this part.
- A way to use the site map information to display the user's current position and give the user the ability to easily move from one place to another. This part takes place through the navigation controls you bind to the `SiteMapDataSource` control, which can include breadcrumb links, lists, menus, and trees.

You can customize or extend each of these ingredients separately. For example, if you want to change the appearance of your navigation controls, you simply need to bind different controls to the SiteMapDataSource. On the other hand, if you want to read site map information from a different type of file or from a different location, you need to change your site map provider.

Figure 13-1 shows how these pieces fit together.

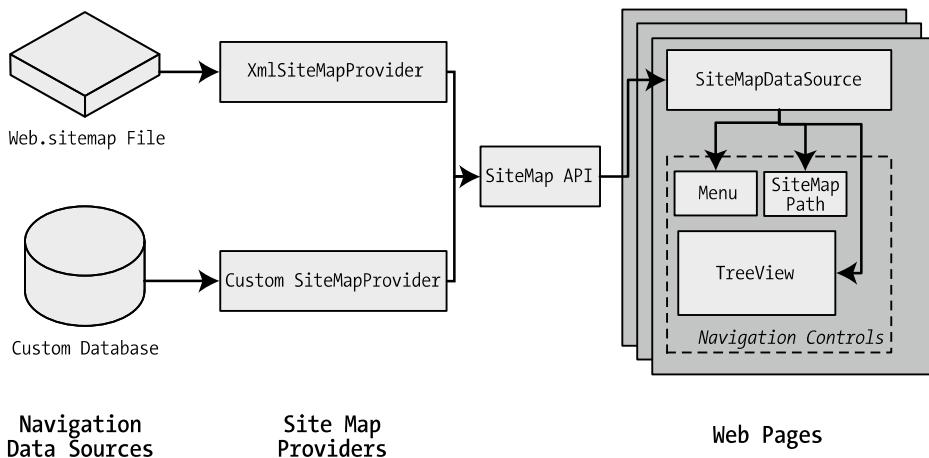


Figure 13-1. ASP.NET navigation with site maps

Defining a Site Map

The starting point in site map-based navigation is the site map provider. ASP.NET ships with a single site map provider, named XmlSiteMapProvider, which is able to retrieve site map information from an XML file. If you want to retrieve a site map from another location or in a custom format, you'll need to create your own site map provider or look for a third-party solution on the Web.

The XmlSiteMapProvider looks for a file named Web.sitemap in the root of the virtual directory. Like all site map providers, the task of the XmlSiteMapProvider is to extract the site map data and create the corresponding SiteMap object. This SiteMap object is then made available to the SiteMapDataSource, which you place on every page that uses navigation. The SiteMapDataSource provides the site map information to navigation controls, which are the final link in the chain.

Tip To simplify the task of adding navigation to your website, you can use master pages, as described in Chapter 12. That way, you simply need to place the SiteMapDataSource and navigation controls on the master page rather than on all the individual pages in your website. You'll use this technique in this chapter.

You can create a site map by using a text editor such as Notepad, or you can create it in Visual Studio by choosing Website > Add New Item and then choosing the Site Map option. Either way, it's up to you to enter all the site map information by hand. The only difference is that if you create it in Visual Studio, the site map will start with a basic structure that consists of three SiteMap nodes.

Before you can fill in the content in your site map file, you need to understand the rules that all ASP.NET site maps must follow. The following sections break these rules down piece by piece.

Note before you begin creating site maps, it helps to have a basic understanding of XML, the format used for the site map file. You should understand what an element is, how to start and end an element, exact capitalization is so important. If you're new to XML, you may find that it helps to refer to Chapter 1 introduction before you read this chapter.

Rule 1: Site Maps Begin with the <siteMap> Element

Every Web.sitemap file begins by declaring the <siteMap> element and ends by closing that element. You place the actual site map information between the start and end tags (where the three dots are shown here):

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
...
</siteMap>
```

The xmlns attribute is required, and must be entered exactly as shown here. This tells ASP.NET that the XML file uses the ASP.NET site map standard.

Rule 2: Each Page Is Represented by a <siteMapNode> Element

So, what does the site map content look like? Essentially, every site map defines an organization of web pages. To insert a page into the site map, you add the <siteMapNode> element with some basic information. Namely, you need to supply the title of the page (which appears in the navigation controls), a description (which you may or may not choose to use), and the URL (the link for the page). You add these three pieces of information by using three attributes—named title, description, and url, as shown here:

```
<siteMapNode title="Home" description="Home" url("~/default.aspx") />
```

Notice that this element ends with the characters `/>`. This indicates it's an *empty element* that represents a start tag and an end tag in one. Empty elements (an XML concept described in Chapter 18) never contain other nodes.

Here's a complete, valid site map file that uses this page to define a website with exactly one page:

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
  <siteMapNode title="Home" description="Home" url "~/default.aspx" />
</siteMap>
```

Notice that the URL for each page begins with the `~/` character sequence. This is quite important. The `~/` characters represent the root folder of your web application. In other words, the URL `~/default.aspx` points to the `default.aspx` file in the root folder. This style of URL isn't required, but it's strongly recommended, because it makes sure you always get the right page. If you were to simply enter the URL `default.aspx` without the `~/` prefix, ASP.NET would look for the `default.aspx` page in the *current* folder. If you have a web application with pages in more than one folder, you'll run into a problem.

For example, if the user browses into a subfolder and clicks the `default.aspx` link, ASP.NET will look for the `default.aspx` page in that subfolder instead of in the root folder. Because the `default.aspx` page isn't in this folder, the navigation attempt will fail with a 404 Not Found error.

Rule 3: A <siteMapNode> Element Can Contain Other <siteMapNode> Elements

Site maps don't consist of simple lists of pages. Instead, they divide pages into groups. To represent this in a site map file, you place one <siteMapNode> inside another. Instead of using the empty element syntax shown previously, you'll need to split your <siteMapNode> element into a start tag and an end tag:

```
<siteMapNode title="Home" description="Home" url="~/default.aspx">
...
</siteMapNode>
```

Now you can slip more nodes inside. Here's an example of a Home group that contains two more pages:

```
<siteMapNode title="Home" description="Home" url="~/default.aspx">
  <siteMapNode title="Products" description="Our products"
    url="~/products.aspx" />
  <siteMapNode title="Hardware" description="Hardware choices"
    url="~/hardware.aspx" />
</siteMapNode>
```

Essentially, this represents the hierarchical group of links shown in Figure 13-2.

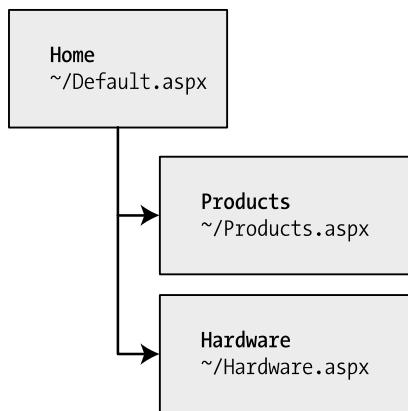


Figure 13-2. Three nodes in a site map

In this case, all three nodes are links. This means the user could surf to one of three pages. However, when you start to create more-complex groups and subgroups, you might want to create nodes that serve only to organize other nodes but aren't links themselves. In this case, just omit the url attribute, as shown here with the Products node:

```
<siteMapNode title="Products" description="Products">
  <siteMapNode title="In Stock" description="Products that are available"
    url="~/inStock.aspx" />
  <siteMapNode title="Not In Stock" description="Products that are on order"
    url="~/outOfStock.aspx" />
</siteMapNode>
```

When you show this part of the site map in a web page, the Products node will appear as ordinary text, not a clickable link.

No limit exists for how many layers deep you can nest groups and subgroups. However, it's a good rule to go just two or three levels deep; otherwise, it may be difficult for users to grasp the hierarchy when they see it in a navigation control. If you find that you need more than two or three levels, you may need to reconsider how you are organizing your pages into groups.

Rule 4: Every Site Map Begins with a Single <siteMapNode>

Another rule applies to all site maps. A site map must always have a single root node. All the other nodes must be contained inside this root-level node.

That means the following is *not* a valid site map, because it contains two top-level nodes:

```
<siteMapNode title="Products" description="Our products"
  url="~/products.aspx" />
<siteMapNode title="Hardware" description="Hardware choices"
  url="~/hardware.aspx" />
```

The following site map is valid, because it has a single top-level node (Home), which contains two more nodes:

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
  <siteMapNode title="Home" description="Home" url="~/default.aspx">
    <siteMapNode title="Products" description="Our products"
      url="~/products.aspx" />
    <siteMapNode title="Hardware" description="Hardware choices"
      url="~/hardware.aspx" />
  </siteMapNode>
</siteMap>
```

As long as you use only one top-level node, you can nest nodes as deep as you want in groups as large or as small as you want.

Rule 5: Duplicate URLs Are Not Allowed

You cannot create two site map nodes with the same URL. This might seem to present a bit of a problem when you want to have the same link in more than one place—and it does. However, it's a requirement because the default SiteMapProvider included with ASP.NET stores nodes in a collection, with each item indexed by its unique URL.

This limitation doesn't prevent you from creating more than one URL with minor differences pointing to the same page. For example, consider the following portion of a site map. These two nodes are acceptable, even though they lead to the same page (products.aspx), because the two URLs have different query string arguments at the end.

```
<siteMapNode title="In Stock" description="Products that are available"
  url="~/products.aspx?stock=1" />
<siteMapNode title="Not In Stock" description="Products that are on order"
  url="~/products.aspx?stock=0" />
```

This approach works well if you have a single page that will display different information, depending on the query string. Using the query string argument, you can add both "versions" of the page to the site map. Chapter 8 describes the query string in more detail.

Note The URL in the site map is not case sensitive.

Seeing a Simple Site Map in Action

A typical site map can be a little overwhelming at first glance. But if you keep the previous five rules in mind, you'll be able to sort out exactly what's taking place.

The following is an example that consists of seven nodes. (Remember, each *node* is either a link to an individual page, or a heading used to organize a group of pages.) The example defines a simple site map for a company named RevoTech.

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">

    <siteMapNode title="Home" description="Home" url("~/default.aspx")>

        <siteMapNode title="Information" description="Learn about our company">
            <siteMapNode title="About Us"
                description="How RevoTech was founded"
                url "~/aboutus.aspx" />
            <siteMapNode title="Investing"
                description="Financial reports and investor analysis"
                url "~/financial.aspx" />
        </siteMapNode>

        <siteMapNode title="Products" description="Learn about our products">
            <siteMapNode title="RevoStock"
                description="Investment software for stock charting"
                url "~/product1.aspx" />
            <siteMapNode title="RevoAnalyze"
                description="Investment software for yield analysis"
                url "~/product2.aspx" />
        </siteMapNode>

    </siteMapNode>

</siteMap>
```

In the following section, you'll bind this site map to the controls in a page, and you'll see its structure emerge.

Binding an Ordinary Page to a Site Map

Once you've defined the Web.sitemap file, you're ready to use it in a page. First, it's a good idea to make sure you've created all the pages that are listed in the site map file, even if you leave them blank. Otherwise, you'll have trouble testing whether the site map navigation actually works.

The next step is to add the SiteMapDataSource control to your page. You can drag and drop it from the Data tab of the Toolbox. It creates a tag like this:

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
```

The SiteMapDataSource control appears as a gray box on your page in Visual Studio, but it's invisible when you run the page.

The last step is to add controls that are linked to the SiteMapDataSource. Although you can use any of the data controls described in Part 3, in practice you'll find that you'll get the results you want only with the three controls that are available in the Navigation tab of the Toolbox. That's because these controls support hierarchical

data (data with multiple nested levels), and the site map is an example of hierarchical data. In any other control, you'll see only a single level of the site map at a time, which is impractical.

These are the three navigation controls:

TreeView: The TreeView displays a "tree" of grouped links that shows your whole site map at a glance.

Menu: The Menu displays a multilevel menu. By default, you'll see only the first level, but other levels pop up (thanks to some nifty JavaScript) when you move the mouse over the subheadings.

SiteMapPath: The SiteMapPath is the simplest navigation control—it displays the full path you need to take through the site map to get to the current page. For example, it might show Home > Products > RevoStock if you're at the product1.aspx page.

Unlike the other navigation controls, the SiteMapPath is useful only for moving up the hierarchy.

To connect a control to the SiteMapDataSource, you simply need to set its DataSourceID property to match the name of the SiteMapDataSource. For example, if you added a TreeView, you should tweak the tag so it looks like this:

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1" />
```

Figure 13-3 shows the result—a tree that displays the structure of the site, as defined in the website. (The text at the bottom is from the current page—in this case, default.aspx, which contains the site map.)

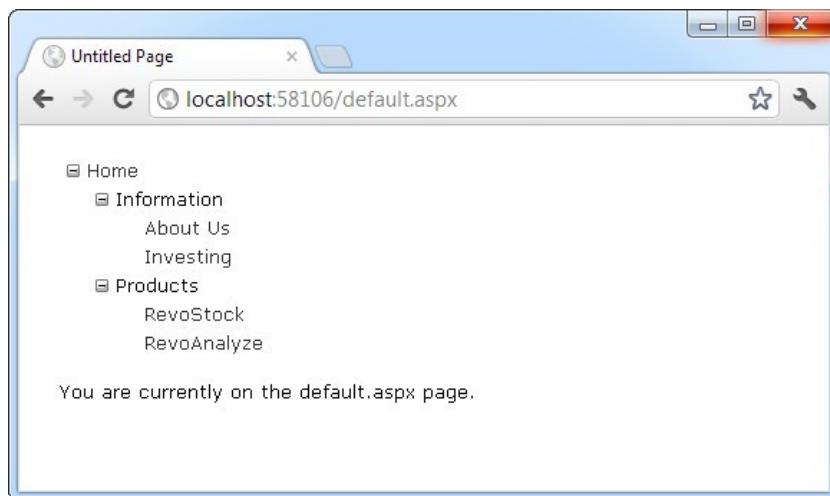


Figure 13-3. A site map in the TreeView

When using the TreeView, the description information doesn't appear immediately. Instead, it's displayed as a tooltip when you hover over an item in the tree.

Best of all, this tree is created automatically. As long as you link it to the SiteMapDataSource control, you don't need to write any code.

When you click one of the nodes in the tree, you'll automatically be taken to the page you defined in the URL. Of course, unless that page also includes a navigation control such as the TreeView, the site map will disappear from sight. The next section shows a better approach.

Binding a Master Page to a Site Map

Website navigation works best when combined with another ASP.NET feature—master pages. That's because you'll usually want to show the same navigation controls on every page. The easiest way to do this is to create a master page that includes the SiteMapDataSource and the navigation controls. You can then reuse this template for every other page on your site.

Here's how you might define a basic structure in your master page that puts navigation controls on the left:

```
<%@ Master Language="C#" AutoEventWireup="true"
  CodeFile="MasterPage.master.cs" Inherits="MasterPage" %>
<html>
<head runat="server">
  <title>Navigation Test</title>
</head>
<body>
<form id="form1" runat="server">
  <table>
    <tr>
      <td style="width: 226px; vertical-align: top;">
        <asp:TreeView ID="TreeView1" runat="server"
          DataSourceID="SiteMapDataSource1" />
      </td>
      <td style="vertical-align: top;">
        <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server" />
      </td>
    </tr>
  </table>
  <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
</form>
</body>
</html>
```

Then create a child with some simple static content. Here's the code for the default.aspx page in this example:

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master"
  AutoEventWireup="true"
  CodeFile="default.aspx.cs" Inherits="_default" Title="Home Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
  runat="Server">
  <br />
  <br />
  You are currently on the default.aspx page (Home).
</asp:Content>
```

In fact, while you're at it, why not create a second page so you can test the navigation between the two pages?

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master"
  AutoEventWireup="true" CodeFile="product1.aspx.cs"
  Inherits="product1" Title="RevoStock Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
  runat="Server">
  <br />
  <br />
  You are currently on the product1.aspx page (RevoStock).
</asp:Content>
```

Now you can jump from one page to another by using the TreeView (see Figure 13-4). The first picture shows the home page as it initially appears, while the second shows the result of clicking the RevoStock link in the TreeView. Because both pages use the same master, and the master page includes the TreeView, the site map always remains visible.

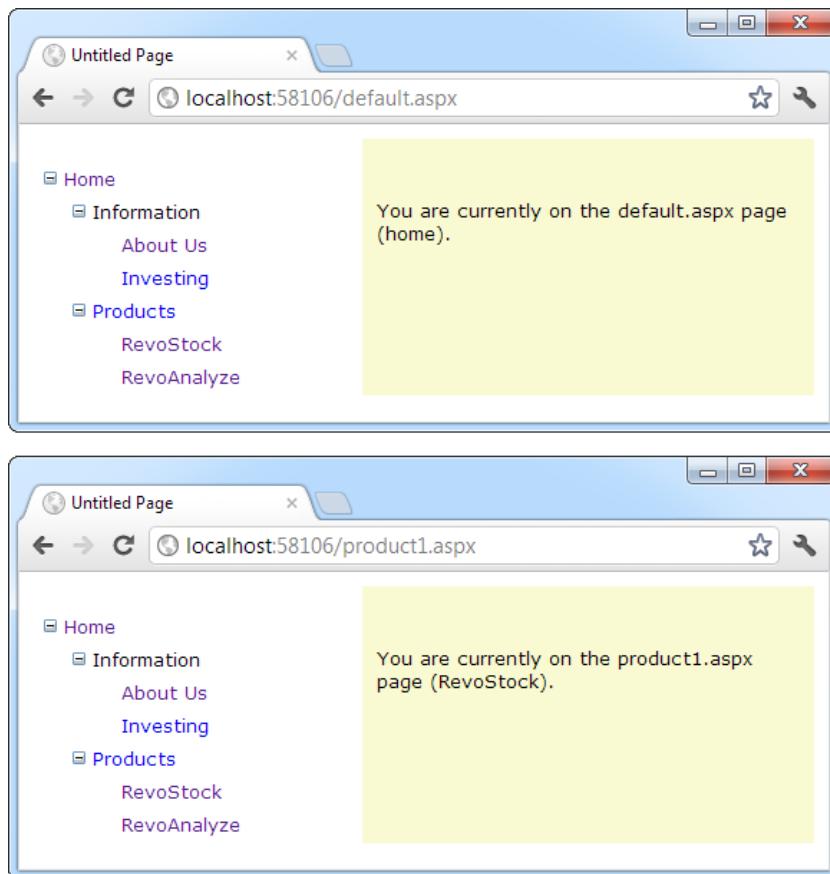


Figure 13-4. Navigating from page to page with the TreeView

You can do a lot more to customize the appearance of your pages and navigation controls. You'll consider these topics in the following sections.

Binding Portions of a Site Map

In the previous example, the TreeView shows the structure of the site map file *exactly*. However, this isn't always what you want. For example, you might not like the way the Home node sticks out because of the XmlSiteMapProvider rule that every site map must begin with a single root.

One way to clean this up is to configure the properties of the SiteMapDataSource. For example, you can set the ShowStartingNode property to false to hide the root node:

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server"
    ShowStartingNode="False" />
```

Figure 13-5 shows the result.

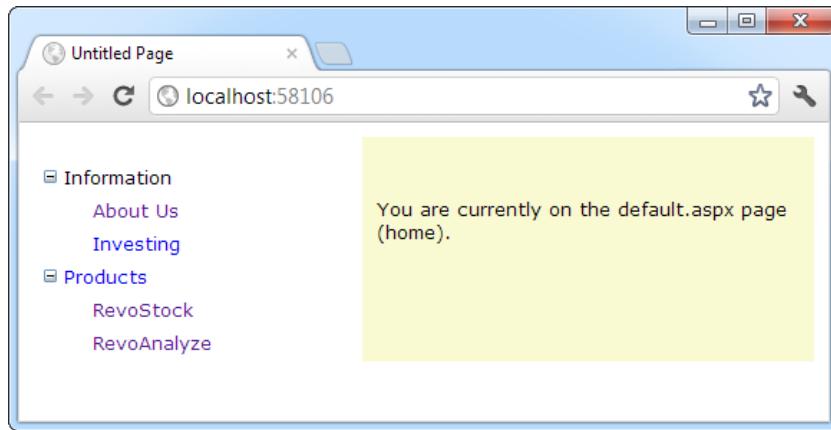


Figure 13-5. A site map without the root node

This example shows how you can hide the root node. Another option is to show just a portion of the complete site map, starting from the current node. For example, you might use a control such as the TreeView to show everything in the hierarchy starting from the current node. A user who wants to move up a level could use another control (such as the SiteMapPath).

Showing Subtrees

By default, the SiteMapDataSource shows a full tree that starts with the root node. However, the SiteMapDataSource has several properties that can help you configure the navigation tree to limit the display to just a specific branch. Typically, this is useful if you have a deeply nested tree. Table 13-1 describes the full set of properties.

Table 13-1. SiteMapDataSource Properties

Property	Description
ShowStartingNode	Set this property to false to hide the first (top-level) node that would otherwise appear in the navigation tree. The default is true.
StartingNodeUrl	Use this property to change the starting node. Set this value to the URL of the node that should be the first node in the navigation tree. This value must match the url attribute in the site map file exactly. For example, if you specify a StartingNodeUrl of "~/home.aspx", then the first node in the tree is the Home node, and you will see nodes only underneath that node.
StartFromCurrentNode	Set this property to true to set the current page as the starting node. The navigation tree will show only pages beneath the current page (which allows the user to move down the hierarchy). If the current page doesn't exist in the site map file, this approach won't work.

(continued)

Table 13-1. (continued)

Property	Description
StartingNodeOffset	Use this property to shift the starting node up or down the hierarchy. It takes an integer that instructs the SiteMapDataSource to move from the starting node down the tree (if the number is positive) or up the tree (if the number is negative). The actual effect depends on how you combine this property with other SiteMapDataSource properties. For example, if StartFromCurrentNode is false, you'll use a positive number to move down the tree, from the starting node toward the current node. If StartFromCurrentNode is true, you'll use a negative number to move up the tree, away from the current node and toward the starting node.

Figuring out these properties can take some work, and you might need to do a bit of experimenting to decide the right combination of SiteMapDataSource settings you want to use. To make matters more interesting, you can use more than one SiteMapDataSource on the same page. This means you could use two navigation controls to show different sections of the site map hierarchy.

Before you can see this in practice, you need to modify the site map file used for the previous few examples into something a little more complex. Currently, the site map has three levels, but only the first level (the Home node) and the third level (the individual pages) have URL links. The second-level groupings (Information and Products) are just used as headings, not links. To get a better feel for how the SiteMapDataSource properties work with multiple navigation levels, modify the Information node as shown here:

```
<siteMapNode title="Information" description="Learn about our company"
  url="~/information.aspx">
```

and change the Products node:

```
<siteMapNode title="Products" description="Learn about our products"
  url="~/products.aspx">
```

Next create the products.aspx and information.aspx pages.

The interesting feature of the Products node is that not only is it a navigable page, but it's a page that has other pages both above it and below it in the navigation hierarchy. This makes it ideal for testing the SiteMapDataSource properties. For example, you can create a SiteMapDataSource that shows only the current page and the pages below it, like this:

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server"
  StartFromCurrentNode="True" />
```

And you can create one that always shows the Information page and the pages underneath it, like this:

```
<asp:SiteMapDataSource ID="SiteMapDataSource2" runat="server"
  StartingNodeUrl="~/information.aspx" />
```

Note For this technique to work, AsP.NET must be able to find a page in the Web.sitemap file that matches the current URL. otherwise, AsP.NET won't know where the current position is, and it won't provide any navigation information to the bound controls.

Now just bind two navigation controls. In this case, one TreeView is linked to each SiteMapDataSource in the markup for the master page:

Pages under the current page:

```
<asp:TreeView ID="TreeView1" runat="server"
    DataSourceID="SiteMapDataSource1" />
<br />
The Information group of pages:<br />
<asp:TreeView ID="TreeView2" runat="server"
    DataSourceID="SiteMapDataSource2" />
```

Figure 13-6 shows the result as you navigate from default.aspx down the tree to products1.aspx. The first TreeView shows the portion of the tree under the current page, and the second TreeView is always fixed on the Information group.

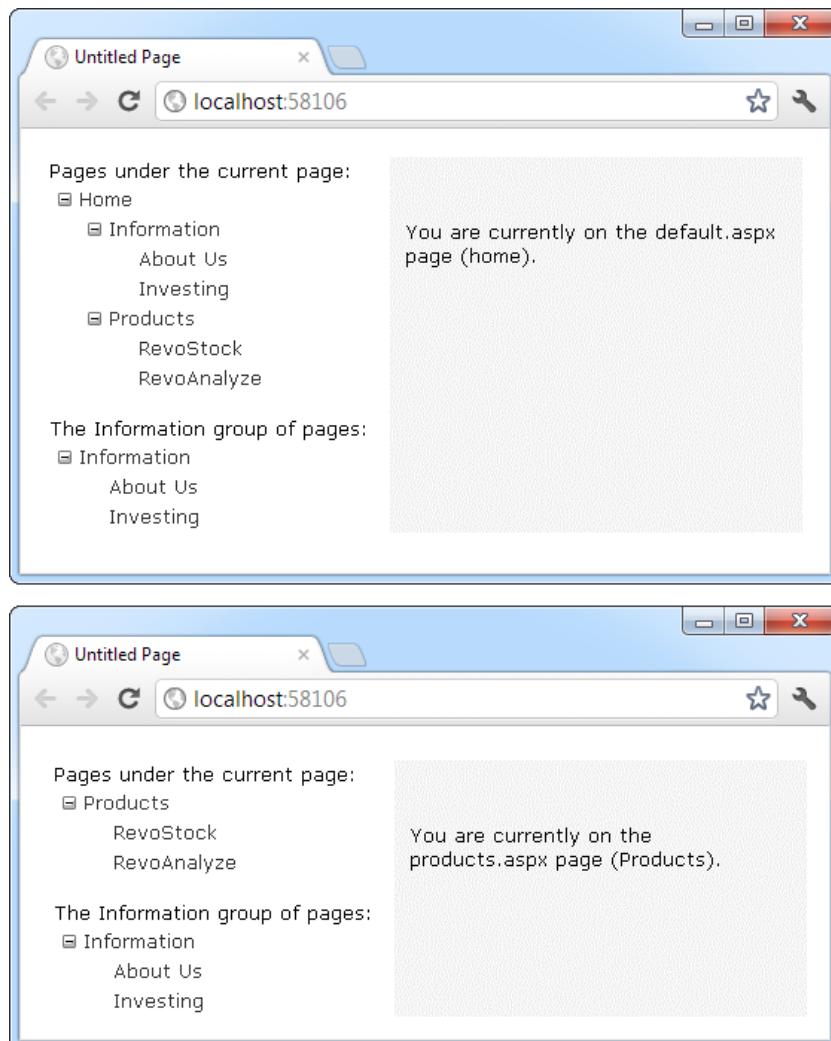


Figure 13-6. Showing portions of the site map

You'll need to get used to the `SiteMapDataSource.StartingNodeOffset` property. It takes an integer that instructs the `SiteMapDataSource` to move that many levels down the tree (if the number is positive) or up the tree (if the number is negative). An important detail that's often misunderstood is that when the `SiteMapDataSource` moves down the tree, it moves *toward* the current node. If it's already at the current node, or your offset takes it beyond the current node, the `SiteMapDataSource` won't know where to go, and you'll end up with a blank navigation control.

To understand how this works, it helps to consider an example. Imagine you're at this location in a website:

Home > Products > Software > Custom > Contact Us

If the `SiteMapDataSource` is starting at the Home node (the default) and you apply a `StartingNodeOffset` of 2, it will move down the tree two levels and bind to the tree of pages that starts at the Software node.

On the other hand, if you're currently at the Products node, you won't see anything. That's because the starting node is Home, and the offset tries to move it down two levels. However, you're only one level deep in the hierarchy. Or, to look at it another way, no node exists between the top node and the current node that's two levels deep.

Now, what happens if you repeat the same test but set the site map provider to begin on another node? Consider what happens if you set `StartFromCurrentNode` to true and surf to the Contact Us page. Once again, you won't see any information, because the site map provider attempts to move two levels down from the current node—Contact Us—and it has nowhere to go. On the other hand, if you set `StartFromCurrentNode` to true and use a `StartingNodeOffset` of -2, the `SiteMapDataSource` will move *up* two levels from Contact Us and bind the subtree starting at Software.

Overall, you won't often use the `StartingNodeOffset` property. However, it can be useful if you have a deeply nested site map and you want to keep the navigation display simple by showing just a few levels up from the current position.

Note All the examples in this section filtered out higher-level nodes than the starting node. For example, you're positioned at the Home > Products > Revostock page, you've seen how to hide the Home and Products. You haven't seen how to filter out lower-level nodes. For example, if you're positioned at the Home page, you always see the full site map, because you don't have a way to limit the number of levels you see below the current node. You have no way to change this behavior with the `siteMapDatasource`; but later, in "The Treeview" section, you'll see that the `Treeview.MaxDatabindDepth` property serves this purpose.

Using Different Site Maps in the Same File

Imagine you want to have a dealer section and an employee section on your website. You might split this into two structures and define them both under different branches in the same file, like this:

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
  <siteMapNode title="Root" description="Root" url="~/default.aspx">
    <siteMapNode title="Dealer Home" description="Dealer Home"
      url="~/default_dealer.aspx">
      ...
    </siteMapNode>
    <siteMapNode title="Employee Home" description="Employee Home"
      url="~/default_employee.aspx">
      ...
    </siteMapNode>
  </siteMapNode>
</siteMap>
```

To bind the SiteMapDataSource to the dealer view (which starts at the Dealer Home page), you simply set the StartingNodeUrl property to “~/default_dealer.aspx”. You can do this programmatically or, more likely, by creating an entirely different master page and implementing it in all your dealer pages. In your employee pages, you set the StartingNodeUrl property to “~/default_employee.aspx”. This way, you’ll show only the pages under the Employee Home branch of the site map.

You can even make your life easier by using the siteMapFile attribute to break a single site map into separate files, like this:

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
  <siteMapNode title="Root" description="Root" url="~/default.aspx">
    <siteMapNode siteMapFile="Dealers.sitemap" />
    <siteMapNode siteMapFile="Employees.sitemap" />
  </siteMapNode>
</siteMap>
```

Even with this technique, you’re still limited to a single site map tree, and it always starts with the Web.sitemap file. But you can manage your site map more easily because you can factor some of its content into separate files.

However, this seemingly nifty technique is greatly limited because the site map provider doesn’t allow duplicate URLs. This means you have no way to reuse the same page in more than one branch of a site map. Although you can try to work around this problem by creating different URLs that are equivalent (for example, by adding query string parameters on the end), this raises more headaches. Sadly, this problem has no solution with the default site map provider that ASP.NET includes.

Working with the SiteMap Class

You aren’t limited to no-code data binding in order to display navigation hierarchies. You can interact with the navigation information programmatically. This allows you to retrieve the current node information and use it to configure details such as the page heading and title. All you need to do is interact with the objects that are readily available through the Page class.

The site map API is remarkably straightforward. To use it, you need to work with two classes from the System.Web namespace. The starting point is the SiteMap class, which provides the static properties CurrentNode (the site map node representing the current page) and RootNode (the root site map node). Both of these properties return a SiteMapNode object. Using the SiteMapNode object, you can retrieve information from the site map, including the title, description, and URL values. You can branch out to consider related nodes by using the navigation properties in Table 13-2.

Table 13-2. SiteMapNode Navigation Properties

Property	Description
ParentNode	Returns the node one level up in the navigation hierarchy, which contains the current node. On the root node, this returns a null reference.
ChildNodes	Provides a collection of all the child nodes. You can check the HasChildNodes property to determine whether child nodes exist.
PreviousSibling	Returns the previous node that’s at the same level (or a null reference if no such node exists).
NextSibling	Returns the next node that’s at the same level (or a null reference if no such node exists).

Note You can also search for nodes by using the methods of the current siteMapProvider object, which is available through the siteMap.Provider static property. For example, the siteMap.Provider.FindsiteMapNode() method allows you to search for a node by its URL.

To see this in action, consider the following code, which configures two labels on a page to show the heading and description information retrieved from the current node:

```
protected void Page_Load(object sender, EventArgs e)
{
    lblHead.Text = SiteMap.CurrentNode.Title;
    lblDescription.Text = SiteMap.CurrentNode.Description;
}
```

If you're using master pages, you could place this code in the code-behind for your master page, so that every page is assigned its title from the site map.

The next example is a little more ambitious. It implements a Next link, which allows the user to traverse an entire set of subnodes. The code checks for the existence of sibling nodes, and if there aren't any in the required position, it simply hides the link:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (SiteMap.CurrentNode.NextSibling != null)
    {
        lnkNext.NavigateUrl = SiteMap.CurrentNode.NextSibling.Url;
        lnkNext.Visible = true;
    }
    else
    {
        lnkNext.Visible = false;
    }
}
```

Figure 13-7 shows the result. The first picture shows the Next link on the product1.aspx page. The second picture shows how this link disappears when you navigate to product2.aspx (either by clicking the Next link or the RevoAnalyze link in the TreeView).

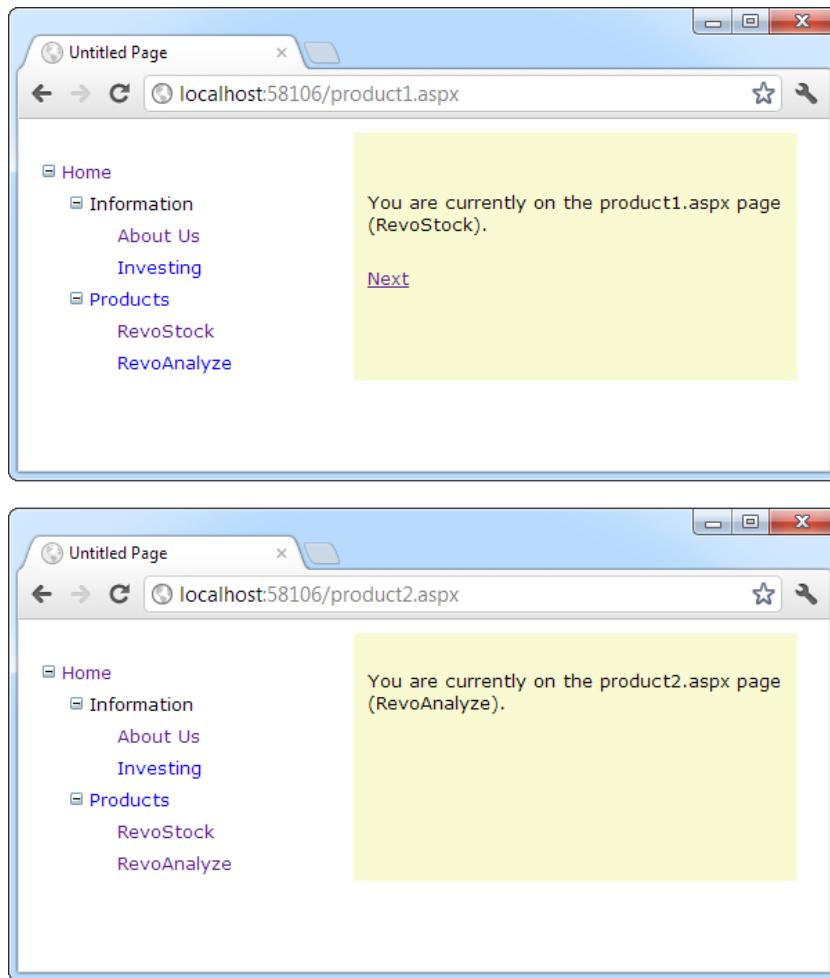


Figure 13-7. Creating a Next page link

URL Mapping and Routing

The site map model is designed around a simple principle: each entry has a separate URL. Although you can distinguish URLs by adding query string arguments, in many websites there is one-to-one correspondence between web forms and site map entries.

When this doesn't suit, ASP.NET has two tools that may be able to help you. The first is URL mapping, which is a clean, no-nonsense way to map one URL to another. The second is URL routing, which is a slightly more elaborate but much more flexible system that performs the same task. URL mapping is an ideal way to deal with "one-off" redirection. For example, mapping is a quick way to deal with old or recently moved pages or to allow extra entry points for a few popular pages. On the other hand, URL routing can serve as the basis for a more sophisticated redirection system that deals with many more pages. For example, you could use it to replace long, complex product page URLs with a simpler syntax and implement that across your entire website. Routing is particularly useful if you want to offer cleaner URLs so that search engines can index your website more easily and comprehensively.

URL Mapping

In some situations, you might want to have several URLs lead to the same page. This might be the case for a number of reasons—maybe you want to implement your logic in one page and use query string arguments but still provide shorter and easier-to-remember URLs to your website users (often called *friendly URLs*). Or maybe you have renamed a page, but you want to keep the old URL functional so it doesn't break user bookmarks. Although web servers sometimes provide this type of functionality, ASP.NET includes its own URL-mapping feature.

The basic idea behind ASP.NET URL mapping is that you map a request URL to a different URL. The mapping rules are stored in the web.config file, and they're applied before any other processing takes place. Of course, for ASP.NET to apply the remapping, it must be processing the request, which means the request URL must use a file type extension that's mapped to ASP.NET (such as .aspx).

You define URL mapping in the <urlMappings> section of the web.config file. You supply two pieces of information—the request URL (as the url attribute) and the new destination URL (mappedUrl). Here's an example:

```
<configuration>
  <system.web>
    <urlMappings enabled="true">
      <add url="~/category.aspx"
           mappedUrl="~/default.aspx?category=default" />
      <add url="~/software.aspx"
           mappedUrl="~/default.aspx?category=software" />
    </urlMappings>
    ...
  </system.web>
</configuration>
```

In order for ASP.NET to make a match, the URL that the browser submits must match the URL specified in the web.config file almost exactly. However, there are two exceptions. First, the matching algorithm isn't case sensitive, so the capitalization of the request URL is always ignored. Second, any query string arguments in the URL are disregarded. Unfortunately, ASP.NET doesn't support advanced matching rules, such as wildcards or regular expressions.

When you use URL mapping, the redirection takes place in the same way as the Server.Transfer() method, which means no round-trip happens and the URL in the browser will still show the original request URL, not the new page. In your code, the Request.Path and Request.QueryString properties reflect the new (mapped) URL. The Request.RawUrl property returns the original, friendly request URL.

This can introduce some complexities if you use it in conjunction with site maps—namely, does the site map provider try to use the original request URL or the destination URL when looking for the current node in the site map? The answer is both. It begins by trying to match the request URL (provided by the Request.RawUrl property), and if no value is found, it then uses the Request.Path property instead. This is the behavior of the XmlSiteMapProvider, so you could change it in a custom provider if desired.

URL Routing

URL routing was originally designed as a core part of ASP.NET MVC, an alternative framework for building web pages that doesn't use the web form features discussed in this book. However, the creators of ASP.NET realized that routing could also help web form developers tame sprawling sites and replace convoluted URLs with cleaner alternatives (which makes it easier for people to type them in and for search engines to index them). For all these reasons, they made the URL-routing feature available to ordinary ASP.NET web form applications.

Note To learn more about AsP.NET MvC, which presents a dramatically different way to think about web pages, check out *Pro ASP.NET MVC 4* (Apress).

Unlike URL mapping, URL routing doesn't take place in the web.config file. Instead, it's implemented using code. Typically, you'll use the Application_Start() method in the global.asax file to register all the routes for your application.

To register a route, you use the RouteTable class from the System.Web.Routing namespace. To make life easier, you can start by importing that namespace:

using System.Web.Routing;

The RouteTable class provides a static property named Routes, which holds a collection of Route objects that are defined for your application. Initially, this collection is empty, but you can create custom routes by calling the MapPageRoute() method, which takes three arguments:

routeName: This is a name that uniquely identifies the route. It can be whatever you want.

routeUrl: This specifies the URL format that browsers will use. Typically, a route URL consists of one or more pieces of variable information, separated by slashes, which are extracted and provided to your code. For example, you might request a product page by using a URL such as /products/4312.

physicalFile: This is the target web form—the place where users will be redirected when they use the route. The information from the original routeUrl will be parsed and made available to this page as a collection through the Page.RouteData property.

Here's an example that adds two routes to a web application when it first starts:

```
protected void Application_Start(object sender, EventArgs e)
{
    RouteTable.Routes.MapPageRoute("product-details",
        "product/{productID}", "~/productInfo.aspx");
    RouteTable.Routes.MapPageRoute("products-in-category",
        "products/category/{categoryID}", "~/products.aspx");
}
```

The route URL can include one or more parameters, represented by a placeholder in curly brackets. For example, the first route shown here includes a parameter named productID. This piece of information will be pulled out of the URL and passed along to the target page.

Here's a URL that uses this route to request a product with the ID FI_00345:

[http://localhost:\[PortNumber\]/Routing/product/FI_00345](http://localhost:[PortNumber]/Routing/product/FI_00345)

The ASP.NET routing infrastructure then redirects the user to the productInfo.aspx page. All the parameters are provided through the Page.RouteData property. Technically, Page.RouteData provides a RouteData object. Its most useful property is the Values collection, which provides all the parameters from the original request, indexed by name.

Here's an example that shows how the productInfo.aspx page can retrieve the requested product ID from the original URL:

```
protected void Page_Load(object sender, EventArgs e)
{
    string productID = (string)Page.RouteData.Values["productID"];
    lblInfo.Text = "You requested " + productID;
}
```

Similarly, the second route in this example accepts URLs in this form:

`http://localhost:[PortNumber]/Routing/products/category/342`

Although you can hard-code this sort of URL, there's a `Page.GetRouteUrl()` helper method that does it for you automatically, avoiding potential mistakes. Here's an example that looks up a route (using its registered name), supplies the parameter information, and then retrieves the corresponding URL.

```
hyperLink.NavigateUrl = Page.GetRouteUrl("product-details", new {productID = "FI_00345" });
```

The slightly strange syntax that passes the parameter information uses a language feature called *anonymous types*. It allows you to supply as few or as many parameters as you want. Technically, the C# compiler automatically creates a class that includes all the parameters you supply and submits that object to the `GetRouteUrl()` method. The final result is a routed URL that points to the FI_00345 product, as shown in the first example.

The SiteMapPath Control

The TreeView shows the available pages, but it doesn't indicate where you're currently positioned. To solve this problem, it's common to use the TreeView in conjunction with the SiteMapPath control. Because the SiteMapPath is always used for displaying navigation information (unlike the TreeView, which can also show other types of data), you don't even need to explicitly link it to the SiteMapDataSource:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server" />
```

The SiteMapPath provides *breadcrumb navigation*, which means it shows the user's current location and allows the user to navigate up the hierarchy to a higher level by using links. Figure 13-8 shows an example with a SiteMapPath control when the user is on the `product1.aspx` page. Using the SiteMapPath control, the user can return to the `default.aspx` page. (If a URL were defined for the `Products` node, you would also be able to click that portion of the path to move to that page.) Once again, the SiteMapPath has been added to the master page, so it appears on all the content pages in your site.

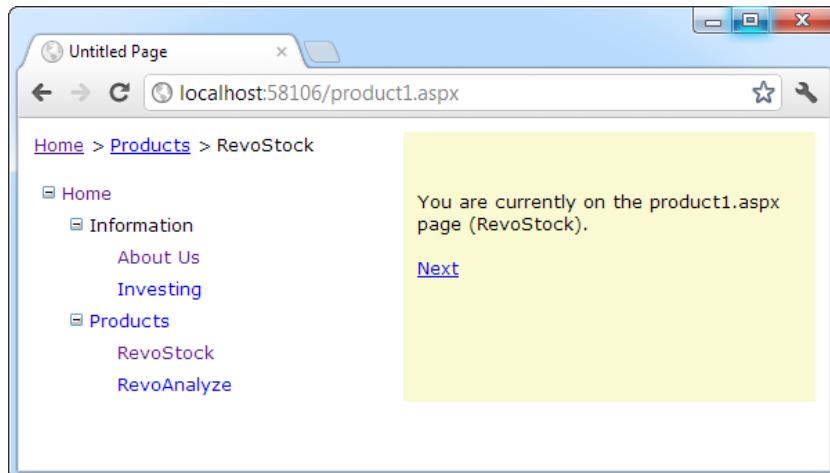


Figure 13-8. Breadcrumb navigation with SiteMapPath

The SiteMapPath control is useful because it provides both an at-a-glance view that shows the current position and a way to move up the hierarchy. However, you always need to combine it with other navigation controls that let the user move down the site map hierarchy.

Customizing the SiteMapPath

The SiteMapPath has a subtle but important difference from other navigation controls such as the TreeView and Menu. Unlike these controls, the SiteMapPath works directly with the ASP.NET navigation model—in other words, it doesn't need to get its data through the SiteMapDataSource. As a result, you can use the SiteMapPath on pages that don't have a SiteMapDataSource, and changing the properties of the SiteMapDataSource won't affect the SiteMapPath. However, the SiteMapPath control provides quite a few properties of its own that you can use for customization. Table 13-3 lists some of its most commonly configured properties.

Table 13-3. SiteMapPath Appearance-Related Properties

Property	Description
ShowToolTips	Set this to false if you don't want the description text to appear when the user hovers over a part of the site map path.
ParentLevelsDisplayed	This sets the maximum number of levels above the current page that will be shown at once. By default, this setting is -1, which means all levels will be shown.
RenderCurrentNodeAsLink	If true, the portion of the page that indicates the current page is turned into a clickable link. By default, this is false because the user is already at the current page.
PathDirection	You have two choices: RootToCurrent (the default) and CurrentToRoot (which reverses the order of levels in the path).
PathSeparator	This indicates the characters that will be placed between each level in the path. The default is the greater-than symbol (>). Another common path separator is the colon (:).

Using SiteMapPath Styles and Templates

For even more control, you can configure the SiteMapPath control with styles or even redefine the controls and HTML with templates. Table 13-4 lists all the styles and templates that are available in the SiteMapPath control; and you'll see how to use both sets of properties in this section.

Table 13-4. SiteMapPath Styles and Templates

Style	Template	Applies To
NodeStyle	NodeTemplate	All parts of the path except the root and current node.
CurrentNodeStyle	CurrentNodeTemplate	The node representing the current page.
RootNodeStyle	RootNodeTemplate	The node representing the root. If the root node is the same as the current node, the current node template or styles are used.
PathSeparatorStyle	PathSeparatorTemplate	The separator between each node.

Styles are easy enough to grasp—they define formatting settings that apply to one part of the SiteMapPath control. Templates are a little trickier, because they rely on data-binding expressions. Essentially, a *template* is a bit of HTML (that you create) that will be shown for a specific part of the SiteMapPath control. For example, if you want to configure how the root node displays in a site map, you could create a SiteMapPath with <RootNodeTemplate> as follows:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server">
<RootNodeTemplate>
<b>Root</b>
</RootNodeTemplate>
</asp:SiteMapPath>
```

This simple template does not use the title and URL information in the root node of the sitemap node. Instead, it simply displays the word *Root* in bold. Clicking the text has no effect.

Usually, you'll use a data-binding expression to retrieve some site map information—chiefly, the description, text, or URL that's defined for the current node in the site map file. Chapter 15 covers data-binding expressions in detail, but this section will present a simple example that shows you all you need to know to use them with the SiteMapPath.

Imagine you want to change how the current node is displayed so that it's shown in italics. To get the name of the current node, you need to write a data-binding expression that retrieves the title. This data-binding expression is bracketed between <%# and %> characters and uses a method named Eval() to retrieve information from a SiteMapNode object that represents a page. Here's what the template looks like:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server">
<CurrentNodeTemplate>
<i><%# Eval("Title") %></i>
</CurrentNodeTemplate>
</asp:SiteMapPath>
```

Data binding also gives you the ability to retrieve other information from the site map node, such as the description. Consider the following example:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server">
<PathSeparatorTemplate>
<asp:Image ID="Image1" ImageUrl="~/arrowright.gif"
runat="server" />
</PathSeparatorTemplate>
<RootNodeTemplate>
<b>Root</b>
</RootNodeTemplate>
<CurrentNodeTemplate>
<%# Eval("Title") %> <br />
<small><i><%# Eval("Description") %></i></small>
</CurrentNodeTemplate>
</asp:SiteMapPath>
```

This SiteMapPath uses several templates. First it uses the PathSeparatorTemplate to define a custom arrow image that's used between each part of the path. This template uses an Image control instead of an ordinary HTML tag because only the Image understands the ~/ characters in the image URL, which represent the application's root folder. If you don't include these characters, the image won't be retrieved successfully if you place your page in a subfolder.

Next the SiteMapPath uses the RootNodeTemplate to supply a fixed string of bold text for the root portion of the site map path. Finally, the CurrentNodeTemplate uses two data-binding expressions to show two pieces of information—both the title of the node and its description (in smaller text, underneath). Figure 13-9 shows the final result.

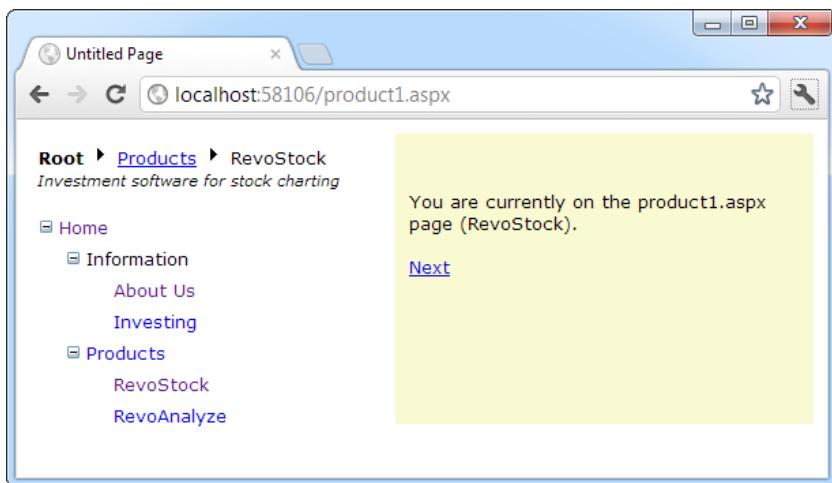


Figure 13-9. A SiteMapPath with templates

Keen eyes will notice that the template-based SiteMapPath not only shows more information but also is more interactive. Now you can click any of the page elements that fall between the root item and the current page. In Figure 13-9, that means you can click Products to move up a level to the products.aspx page.

Interestingly, the templates in the SiteMapPath don't contain any elements that provide these links. Instead, the SiteMapPath automatically determines what items should be clickable (by checking if they're linked to a page in the site map). If an item should be clickable, the SiteMapPath wraps the entire CurrentNodeTemplate for that item inside a link.

If you don't want links (or you want to link in a different way, or with a different control), you can change this behavior. The trick is to modify the NodeTemplate. You'll learn how to do this in the next section.

Adding Custom Site Map Information

In the site maps you've seen so far, the only information that's provided for a node is the title, description, and URL. This is the bare minimum of information you'll want to use. However, the schema for the XML site map is open, which means you're free to insert custom attributes with your own data.

You might want to insert additional node data for a number of reasons. This additional information might be descriptive information that you intend to display, or contextual information that describes how the link should work. For example, you could add an attribute specifying that the link should open in a new window. The only catch is that it's up to you to act on the information later. In other words, you need to configure your user interface so it uses this extra information.

For example, the following code shows a site map that uses a target attribute to indicate the frame where the link should open. In this example, one link is set with a target of _blank so it will open in a new browser window:

```
<siteMapNode title="RevoStock"
    description="Investment software for stock charting"
    url("~/product1.aspx" target="_blank" />
```

Now in your code, you have several options. If you're using a template in your navigation control, you can bind directly to the new attribute. Here's an example with the SiteMapPath from the previous section:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server" Width="264px" Font-Size="10pt">
  <NodeTemplate>
    <a href='<%# Eval("Url") %>' target='<%# Eval("[target]") %>'>
      <%# Eval("Title") %>
    </a>
  </NodeTemplate>
</asp:SiteMapPath>
```

This creates a link that uses the node URL (as usual) but also uses the target information. There's a slightly unusual detail in this example—the square brackets around the word [target]. You need to use this syntax to look up any custom attribute you add to the Web.sitemap file. That's because this value can't be retrieved directly from a property of the SiteMapNode class—instead, you need to use the SiteMapNode indexer to look it up by name.

If your navigation control doesn't support templates, you'll need to find another approach. For example, the TreeView doesn't support templates, but it fires a TreeNodeDataBound event each time an item is bound to the tree. You can react to this event to customize the current item. To apply the new target, use this code:

```
protected void TreeView1_TreeNodeDataBound(object sender, TreeNodeEventArgs e)
{
  SiteMapNode node = (SiteMapNode)e.Node.DataItem;
  e.Node.Target = node["target"];
}
```

As in the template, you can't retrieve the custom attribute from a strongly typed SiteMapNode property. Instead, you use the SiteMapNode indexer to retrieve it by name.

The TreeView Control

You've already seen the TreeView at work for displaying navigation information. As you've learned, the TreeView can show a portion of the full site map or the entire site map. Each node becomes a link that, when clicked, takes the user to the new page. If you hover over a link, you'll see the corresponding description information appear in a tooltip.

In the following sections, you'll learn how to change the appearance of the TreeView. In later chapters, you'll learn how to use the TreeView for other tasks, such as displaying data from a database.

Note The Treeview is one of the most impressive controls in AsP.NET. Not only does it allow you to show maps, but it also supports showing information from a database and filling portions of the tree on demand (without refreshing the entire page). but most important, it supports a wide range of styles that can transform appearance.

TreeView Properties

The TreeView has a slew of properties that let you change how it's displayed on the page. One of the most important properties is ImageSet, which lets you choose a predefined set of node icons. (Each set includes three icons: one for collapsed nodes, one for expanded nodes, and one for nodes that have no children and therefore can't be expanded or collapsed.) The TreeView offers 16 possible ImageSet values, which are represented by the TreeViewImageSet enumeration.

For example, Figure 13-10 shows the same RevoStock navigation page you considered earlier, but this time with an ImageSet value of TreeViewImageSet.Faq. The result is help-style icons that show a question mark (for nodes that have no children) or a question mark superimposed over a folder (for nodes that do contain children).

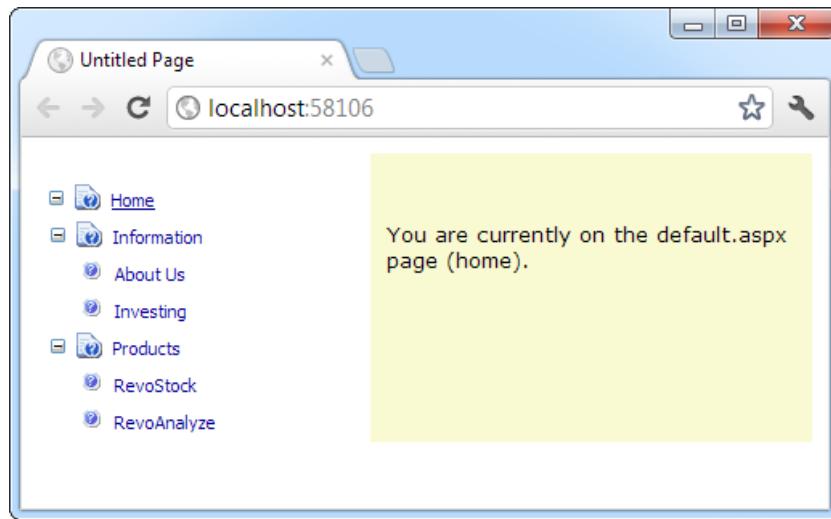


Figure 13-10. A TreeView with fancy node icons

You'll notice that this TreeView makes one more change. It removes the indentation between different levels of nodes, so all the sitemap entries fit in the same narrow column, no matter how many levels deep they are. This is accomplished by setting the NodeIndent property of the TreeView to 0.

Here's the complete TreeView markup:

```
<asp:TreeView ID="TreeView1" runat="server"
    DataSourceID="SiteMapDataSource1" ImageSet="Faq" NodeIndent="0" >
</asp:TreeView>
```

The TreeViewImageSet values are useful if you don't have a good set of images handy. Figure 13-11 shows a page with several TreeViews, each of which represents one of the options in the Auto Format window.

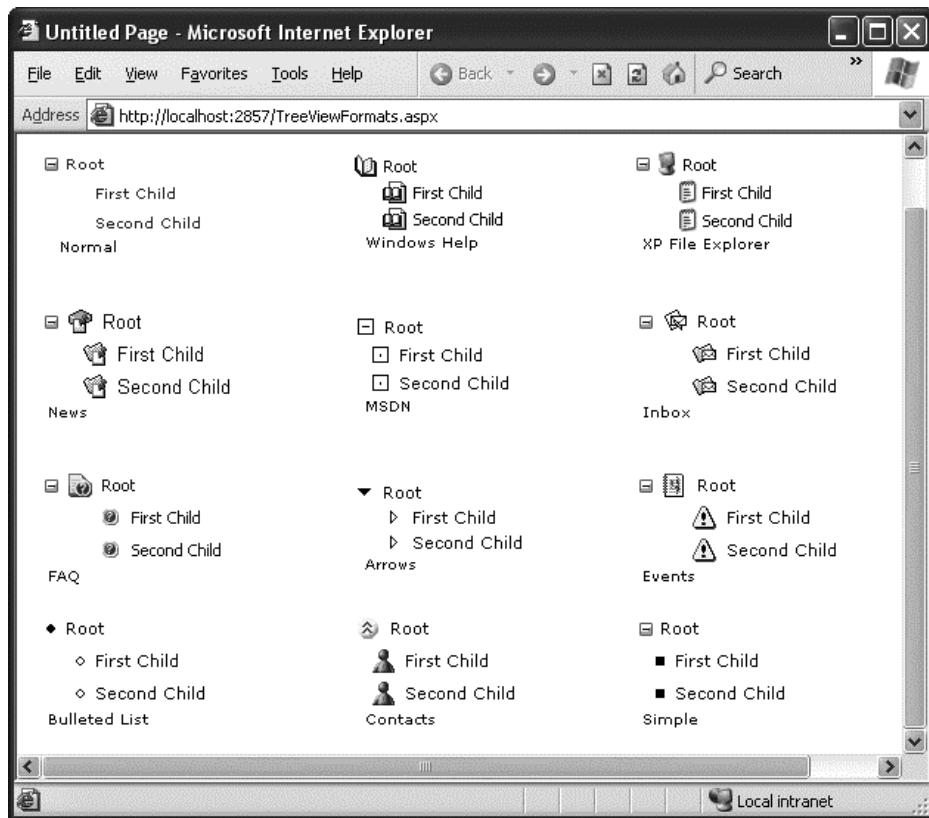


Figure 13-11. Different looks for a TreeView

Although the ImageSet and NodeIndent options can have a dramatic effect on their own, they aren't the only choices when configuring a TreeView. Table 13-5 lists some of the most useful properties of the TreeView.

Table 13-5. Useful TreeView Properties

Property	Description
MaxDataBindDepth	Determines how many levels the TreeView will show. By default, MaxDataBindDepth is -1, and you'll see the entire tree. However, if you use a value such as 2, you'll see only two levels under the starting node. This can help you pare down the display of long, multileveled site maps.
ExpandDepth	Lets you specify how many levels of nodes will be visible at first. If you use 0, the TreeView begins completely closed. If you use 1, only the first level is expanded, and so on. By default, ExpandDepth is set to the constant FullyExpand (-1), which means the tree is fully expanded and all the nodes are visible on the page.
NodeIndent	Sets the number of pixels between each level of nodes in the TreeView. Set this to 0 to create a nonindented TreeView, which saves space. A nonindented TreeView allows you to emulate an in-place menu (see, for example, Figure 13-12).

(continued)

Table 13-5. (continued)

Property	Description
ImageSet	Lets you use a predefined collection of node images for collapsed, expanded, and nonexpandable nodes. You specify one of the values in the TreeViewImageSet enumeration. You can override any node images you want to change by setting the CollapseImageUrl, ExpandImageUrl, and NoExpandImageUrl properties.
CollapseImageUrl, ExpandImageUrl, and NoExpandImageUrl	Sets the pictures that are shown next to nodes for collapsed nodes (CollapseImageUrl) and expanded nodes (ExpandImageUrl). The NoExpandImageUrl is used if the node doesn't have any children. If you don't want to create your own custom node images, you can use the ImageSet property instead to use one of several built-in image collections.
NodeWrap	Lets a node text-wrap over more than one line when set to true.
ShowExpandCollapse	Hides the expand/collapse boxes when set to false. This isn't recommended, because the user won't have a way to expand or collapse a level without clicking it (which causes the browser to navigate to the page).
ShowLines	Adds lines that connect every node when set to true.
ShowCheckboxes	Shows a check box next to every node when set to true. This isn't terribly useful for site maps, but it is useful with other types of trees.

Properties give you a fair bit of customizing power, but one of the most interesting formatting features comes from TreeView styles, which are described in the next section.

TreeView Styles

Styles are represented by the TreeNodeStyle class, which derives from the more conventional Style class. As with other rich controls, the styles give you options to set background and foreground colors, fonts, and borders. Additionally, the TreeNodeStyle class adds the node-specific style properties shown in Table 13-6. These properties deal with the node image and the spacing around a node.

Table 13-6. TreeNodeStyle-Added Properties

Property	Description
ImageUrl	The URL for the image shown next to the node.
NodeSpacing	The space (in pixels) between the current node and the node above and below.
VerticalPadding	The space (in pixels) between the top and bottom of the node text and border around the text.
HorizontalPadding	The space (in pixels) between the left and right of the node text and border around the text.
ChildNodesPadding	The space (in pixels) between the last child node of an expanded parent node and the following node (for example, between the Investing and Products nodes in Figure 13-10).

Because a TreeView is rendered using an HTML table, you can set the padding of various elements to control the spacing around text, between nodes, and so on. One other property that comes into play is TreeView.NodeIndent, which sets the number of pixels of indentation (from the left) in each subsequent level of the tree hierarchy. Figure 13-12 shows how these settings apply to a single node.

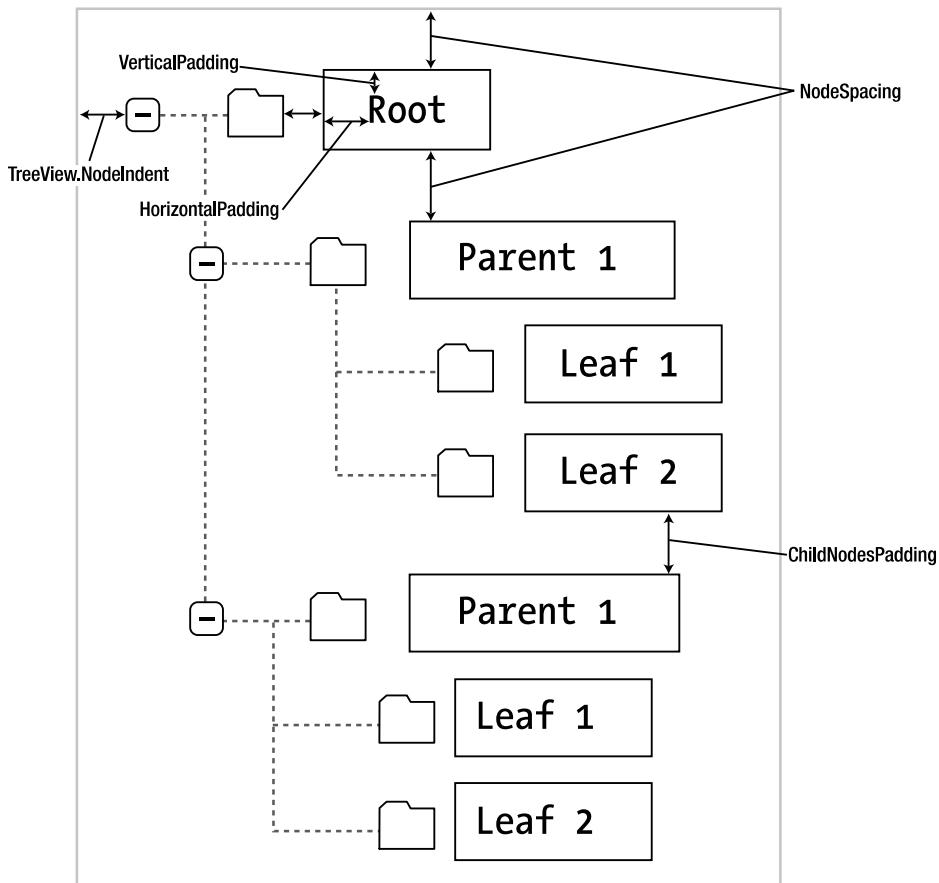


Figure 13-12. Node spacing

Clearly, styles give you a lot of control over how different nodes are displayed. To apply a simple TreeView makeover, and to use the same style settings for each node in the TreeView, you apply style settings through the TreeView.NodeStyle property. You can do this directly in the control tag or by using the Properties window.

For example, here's a TreeView that applies a custom font, font size, text color, padding, and spacing:

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1">
  <NodeStyle Font-Names="Tahoma" Font-Size="10pt" ForeColor="Blue"
    HorizontalPadding="5px" NodeSpacing="0px" VerticalPadding="0px" />
</asp:TreeView>
```

Usually, this approach doesn't provide enough fine-tuning. Instead, you'll want to tweak a specific part of the tree. In this case, you need to find the style object that applies to the appropriate part of the tree, as explained in the following two sections.

Applying Styles to Node Types

The TreeView allows you to individually control the styles for types of nodes—for example, root nodes, nodes that contain other nodes, selected nodes, and so on. Table 13-7 lists different TreeView styles and explains what nodes they affect.

Table 13-7. TreeView Style Properties

Property	Description
NodeStyle	Applies to all nodes. The other styles may override some or all of the details that are specified in the NodeStyle property.
RootNodeStyle	Applies only to the first-level (root) node.
ParentNodeStyle	Applies to any node that contains other nodes, except root nodes.
LeafNodeStyle	Applies to any node that doesn't contain child nodes and isn't a root node.
SelectedNodeStyle	Applies to the currently selected node.
HoverNodeStyle	Applies to the node that the user is hovering over with the mouse. These settings are applied only in up-level clients that support the necessary dynamic script.

Here's a sample TreeView that first defines a few standard style characteristics by using the NodeStyle property, and then fine-tunes different sections of the tree by using the properties from Table 13-7:

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1">
  <NodeStyle Font-Names="Tahoma" Font-Size="10pt" ForeColor="Blue"
    HorizontalPadding="5px" NodeSpacing="0px" VerticalPadding="0px" />
  <ParentNodeStyle Font-Bold="False" />
  <HoverNodeStyle Font-Underline="True" ForeColor="#5555DD" />
  <SelectedNodeStyle Font-Underline="True" ForeColor="#5555DD" />
</asp:TreeView>
```

Styles are listed in Table 13-7 in order of most general to most specific. This means the SelectedNodeStyle settings override any conflicting settings in a RootNodeStyle, for example. (If you don't want a node to be selectable, set the TreeNode.SelectAction to None.) However, the RootNodeStyle, ParentNodeStyle, and LeafNodeStyle settings never conflict, because the definitions for root, parent, and leaf nodes are mutually exclusive. You can't have a node that is simultaneously a parent and a root node, for example—the TreeView simply designates this as a root node.

Applying Styles to Node Levels

Being able to apply styles to different types of nodes is interesting, but often a more useful feature is being able to apply styles based on the node *level*. That's because many trees use a rigid hierarchy. (For example, the first level of nodes represents categories, the second level represents products, the third represents orders, and so on.) In this case, it's not so important to determine whether a node has children. Instead, it's important to determine the node's depth.

The only problem is that a TreeView can have a theoretically unlimited number of node levels. Thus, it doesn't make sense to expose properties such as FirstLevelStyle, SecondLevelStyle, and so on. Instead, the TreeView has a LevelStyles collection that can have as many entries as you want. The level is inferred from the position of the style in the collection, so the first entry is considered the root level, the second entry is the second node level, and so on. For this system to work, you must follow the same order, and you must include an empty style placeholder if you want to skip a level without changing the formatting.

For example, here's a TreeView that differentiates levels by applying different amounts of spacing and different fonts:

```
<asp:TreeView runat="server" HoverNodeStyle-Font-Underline="True"
ShowExpandCollapse="False" NodeIndent="3" DataSourceID="SiteMapDataSource1">
<LevelStyles>
<asp:TreeNodeStyle ChildNodesPadding="10" Font-Bold="True" Font-Size="12pt"
ForeColor="DarkGreen"/>
<asp:TreeNodeStyle ChildNodesPadding="5" Font-Bold="True" Font-Size="10pt" />
<asp:TreeNodeStyle ChildNodesPadding="5" Font-UnderLine="True"
Font-Size="10pt" />
</LevelStyles>
</asp:TreeView>
```

If you apply this to the category and product list shown in earlier examples, you'll see a page like the one shown in Figure 13-13.

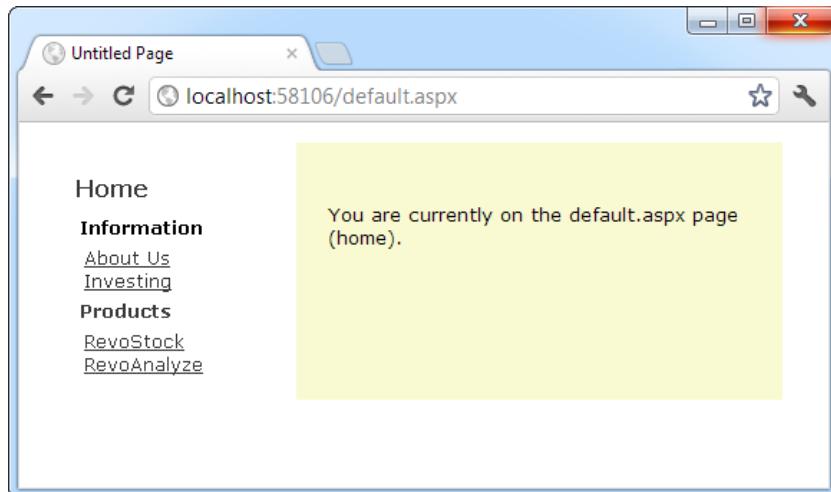


Figure 13-13. A TreeView with styles

treeview auto Format

Using the right combination of styles and images can dramatically transform your Treeview. However, for those less artistically inclined, it's comforting to know that Microsoft has made many classic designs available through the Treeview's Auto Format feature.

To use it, start by selecting the Treeview on the design surface. Then click the arrow icon that appears to the top-right corner of the Treeview to show its smart tag. In the smart tag, click the Auto Format link to show the Auto Format dialog box. In the Auto Format dialog box, you can pick from a variety of present formats, each with a small preview. Click Apply to try the format out on your Treeview, Cancel to back out, and then make it official and return to Visual Studio.

The different formats correspond loosely to the different Treeviewimageset values. However, the results aren't quite that simple. When you pick a Treeview format, Visual Studio sets the imageset property and applies a few matching style settings, to help you get that perfect final look.

The Menu Control

The Menu control is another rich control that supports hierarchical data. Like the TreeView, you can bind the Menu control to a data source, or you can use MenuItem objects to fill it by hand.

To try the Menu control, remove the TreeView from your master page, and add the following Menu control tag:

```
<asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1" />
```

Notice that this doesn't configure any properties—it uses the default appearance. The only step you need to perform is setting the DataSourceID property to link the menu to the site map information.

When the Menu first appears, you'll see only the starting node, with an arrow next to it. When you move your mouse over the starting node, the next level of nodes will pop into display. You can continue this process to drill down as many levels as you want, until you find the page you want to click (see Figure 13-14). If you click a menu item, you'll be transported to the corresponding page, just as you are when you click a node in the TreeView. But unlike the TreeView, each time you click your way to a new page, the menu collapses itself back to its original appearance. It doesn't expand to show the current page.

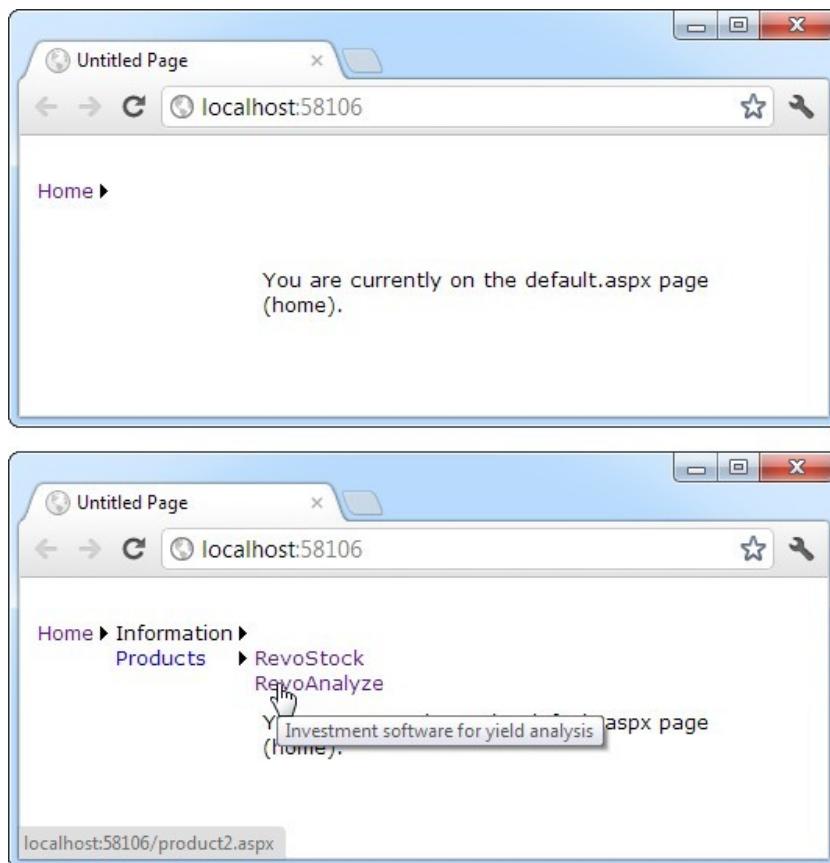


Figure 13-14. Navigating through the menu

Overall, the Menu and TreeView controls expose strikingly similar programming models, even though they render themselves quite differently. They also have a similar style-based formatting model. But a few noteworthy differences exist:

- The Menu displays a single submenu. The TreeView can expand an arbitrary number of node branches at a time.
- The Menu displays a root level of links in the page. All other items are displayed using fly-out menus that appear over any other content on the page. The TreeView shows all its items inline in the page.
- The Menu supports templates. The TreeView does not. (Menu templates are discussed later in this section.)
- The TreeView supports check boxes for any node. The Menu does not.
- The Menu supports horizontal and vertical layouts, depending on the Orientation property. The TreeView supports only vertical layout.

Menu Styles

The Menu control provides an overwhelming number of styles. Like the TreeView, the Menu adds a custom style class, which is named MenuItemStyle. This style adds spacing properties such as ItemSpacing, HorizontalPadding, and VerticalPadding. However, you can't set menu item images through the style, because it doesn't have an ImageUrl property.

Much like the TreeView, the Menu supports defining different menu styles for different menu levels. However, the key distinction that the Menu control encourages you to adopt is between *static* items (the root-level items that are displayed in the page when it's first generated) and *dynamic* items (the items in fly-out menus that are added when the user moves the mouse over a portion of the menu). Most websites have a definite difference in the styling of these two elements. To support this, the Menu class defines two parallel sets of styles, one that applies to static items and one that applies to dynamic items, as shown in Table 13-8.

Table 13-8. Menu Styles

Static Style	Dynamic Style	Description
StaticMenuItemStyle	DynamicMenuItemStyle	Sets the appearance of the overall "box" in which all the menu items appear. In the case of StaticMenuItemStyle, this box appears on the page, and with DynamicMenuItemStyle, it appears as a pop-up.
StaticSelectedStyle	DynamicSelectedStyle	Sets the appearance of individual menu items.
StaticHoverStyle	DynamicHoverStyle	Sets the appearance of the selected item. Note that the selected item isn't the item that's currently being hovered over; it's the item that was previously clicked (and that triggered the last postback).
		Sets the appearance of the item that the user is hovering over with the mouse.

Along with these styles, you can set level-specific styles so that each level of menu and submenu is different. You do this by using the LevelMenuItemStyles collection, which works like the TreeView.LevelStyles collection discussed earlier. The position of your style in the collection determines whether the menu uses it for the first level of items, the second level, the third, and so on. You can also use the LevelSelectedStyles collection to set level-specific formatting for selected items.

It might seem as if you have to do a fair bit of unnecessary work when separating dynamic and static styles. The reason for this model becomes obvious when you consider another remarkable feature of the Menu control—it allows you to choose the number of static levels. By default, only one static level exists, and everything else is displayed as a fly-out menu when the user hovers over the corresponding parent. But you can set the Menu.StaticDisplayLevels property to change all that. If you set it to 2, for example, the first two levels of the menu will be rendered in the page by using the static styles. (You can control the indentation of each level by using the StaticSubMenuIndent property.)

Figure 13-15 shows the menu with StaticDisplayLevels set to 2 (and some styles applied through the Auto Format link). Each menu item will still be highlighted when you hover over it, as in a nonstatic menu, and selection will also work the same way as it does in the nonstatic menu.

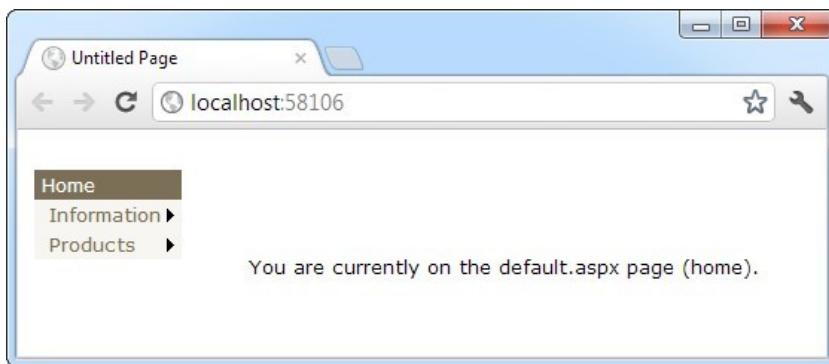


Figure 13-15. A menu with two static levels

If you use a menu for navigation, every time you click your way to a new page, the menu returns to its original appearance, showing all static levels but hiding all dynamic levels.

Tip The Menu control exposes many more top-level properties for tweaking specific rendering aspects. For example, you can set the delay before a pop-up menu disappears (DisappearAfter), the default images used for expansion icons and separators, the scrolling behavior (which kicks into gear when the browser window is too small to fit a pop-up menu), and much more. For more information, you can read the Menu control reference at <http://msdn.microsoft.com/library/system.web.ui.webcontrols.menu.aspx>.

Menu Templates

The Menu control also supports templates through the `StaticItemTemplate` and `DynamicItemTemplate` properties. These templates determine the HTML that's rendered for each menu item, giving you complete control.

You've already seen how to create templates for the `SiteMapPath`, but the process of creating templates for the `Menu` is a bit different. Whereas each node in the `SiteMapPath` is bound directly to a `SiteMapNode` object, the `Menu` is bound to something else: a dedicated `MenuItem` object.

This subtle quirk can complicate life. For one thing, you can't rely on properties such as `Title`, `Description`, and `Url`, which are provided by the `SiteMapNode` object. Instead, you need to use the `MenuItem.Text` property to get the information you need to display, as shown here:

```
<asp:Menu ID="Menu1" runat="server">
<StaticItemTemplate>
  <%# Eval("Text") %>
</StaticItemTemplate>
</asp:Menu>
```

One reason you might want to use the template features of the `Menu` is to show multiple pieces of information in a menu item. For example, you might want to show both the title *and* the description from the `SiteMapNode` for this item (rather than just the title). Unfortunately, that's not as easy as it is with the `SiteMapPath`. Once again, the problem is that the `Menu` binds directly to `MenuItem` objects, not the `SiteMapNode` objects, and `MenuItem` objects just don't provide the information you need.

If you're really desperate, there is a workaround using an advanced data-binding technique. Rather than binding to a property of the MenuItem object, you can bind to a custom method that you create in your page class. This custom method can then include the code that's needed to get the correct SiteMapNode object (based on the current URL) and provide the extra information you need. In a perfect world, this extra work wouldn't be necessary, but unfortunately, it's the simplest workaround in this situation.

For example, consider the more descriptive menu items that are shown in Figure 13-16.

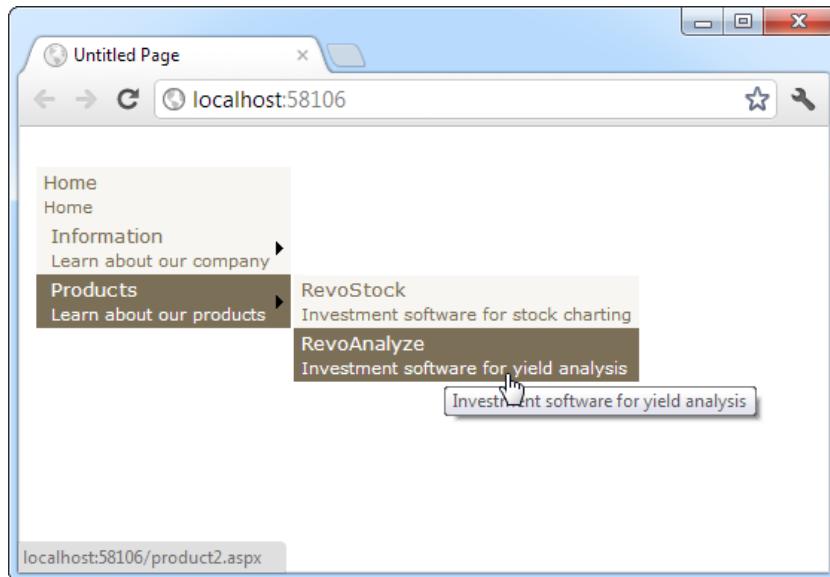


Figure 13-16. Showing node descriptions in a menu

To create this example, you need to build a template that uses two types of data-binding expressions. The first type simply gets the MenuItem text (which is the page title). You already know how to write this sort of data-binding expression:

```
<%# Eval("Text") %>
```

The second type of data-binding expression is more sophisticated. It uses a custom method named `GetDescriptionFromTitle()`, which you need to create. This method takes the page title information and returns something more interesting—in this case, the full description for that item:

```
<%# GetDescriptionFromTitle(((MenuItem)Container.DataItem).Text) %>
```

Sadly, the `Eval()` method can't help you out with this sort of data-binding expression. Instead, you need to explicitly grab the data object (using `Container.DataItem`), cast it to the appropriate type (`MenuItem`), and then retrieve the right property (`Text`). This gets the same page title as in the previous data-binding expression, but it allows you to pass it to the `GetDescriptionFromTitle()` method.

Here's the full template that uses both types of data-binding expressions to show the top level of static menu items and the second level of pop-up (dynamic) items:

```
<asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1">
<StaticItemTemplate>
  <%# Eval("Text") %><br />
```

```

<small>
<%# GetDescriptionFromTitle(((MenuItem)Container.DataItem).Text)%>
</small>
</StaticItemTemplate>
<DynamicItemTemplate>
<%# Eval("Text")%><br />
<small>
<%# GetDescriptionFromTitle(((MenuItem)Container.DataItem).Text)%>
</small>
</DynamicItemTemplate>
</asp:Menu>

```

The next step is to create the `GetDescriptionFromTitle()` method in the code for your page class. This method belongs in the page that has the `Menu` control, which, in this example, is the master page. The `GetDescriptionFromTitle()` method must also have protected (or public) accessibility, so that ASP.NET can call it during the data-binding process:

```

protected string GetDescriptionFromTitle(string title)
{...}

```

The tricky part is filling in the code you need. In this example, two custom methods are involved. In order to find the node it needs, `GetDescriptionFromTitle()` calls another method, named `SearchNodes()`. The `SearchNodes()` method calls itself several times to perform a recursive search through the whole hierarchy of nodes. It ends its search only when it finds a matching node, which it returns to `GetDescriptionFromTitle()`. Finally, `GetDescriptionFromTitle()` extracts the description information (and anything else you're interested in).

Here's the complete code that makes this example work:

```

protected string GetDescriptionFromTitle(string title)
{
    // This assumes there's only one node with this title.
    SiteMapNode startingNode = SiteMap.RootNode;
    SiteMapNode matchNode = SearchNodes(startingNode, title);
    if (matchNode == null)
    {
        return null;
    }
    else
    {
        return matchNode.Description;
    }
}

private SiteMapNode SearchNodes(SiteMapNode node, string title)
{
    if (node.Title == title)
    {
        return node;
    }
    else
    {
        // Perform recursive search.
        foreach (SiteMapNode child in node.ChildNodes)

```

```
{  
    SiteMapNode matchNode = SearchNodes(child, title);  
    // Was a match found?  
    // If so, return it.  
    if (matchNode != null) return matchNode;  
}  
// All the nodes were examined, but no match was found.  
return null;  
}  
}
```

Once you've finished this heavy lifting, you can use the `GetDescriptionFromTitle()` method in a template to get the additional information you need.

The Last Word

In this chapter, you explored the new navigation model and learned how to define site maps and bind the navigation data. You then considered three controls that are specifically designed for navigation data: the `SiteMapPath`, `TreeView`, and `Menu`. Using these controls, you can add remarkably rich site maps to your websites with very little coding. But before you begin, make sure you've finalized the structure of your website. Only then will you be able to create the perfect site map and choose the best ways to present the site map information in the navigation controls.



Error Handling, Logging, and Tracing

No software can run free from error, and ASP.NET applications are no exception. Sooner or later, your code will be interrupted by a programming mistake, invalid data, unexpected circumstances, or even hardware failure. Novice programmers spend sleepless nights worrying about errors. Professional developers recognize that bugs are an inherent part of software applications and so they code defensively, testing assumptions and writing error-handling code to deal with the unexpected.

In this chapter, you'll learn how to use structured exception handling to defend your ASP.NET applications against common errors. You'll also take a look at page tracing, which allows you to see diagnostic information about ASP.NET pages and may help you solve mysterious issues.

Avoiding Common Errors

Errors can occur in a variety of situations. Some of the most common causes of errors include attempts to divide by zero (usually caused by invalid input or missing information) and attempts to connect to a limited resource such as a file or a database (which can fail if the file doesn't exist, the database connection times out, or the code has insufficient security credentials).

One infamous type of error is the *null reference exception*, which usually occurs when a program attempts to use an uninitialized object. As a .NET programmer, you'll quickly learn to recognize and resolve this common but annoying mistake. The following code example shows the problem in action, with two SqlConnection objects that represent database connections:

```
// Define a variable named conOne and create the object.  
private SqlConnection conOne = new SqlConnection();  
  
// Define a variable named conTwo, but don't create the object.  
private SqlConnection conTwo;  
  
public void cmdDoSomething_Click(object sender, EventArgs e)  
{  
    // This works, because the object has been created  
    // with the new keyword.  
    conOne.ConnectionString = "...";  
    ...  
  
    // The following statement will fail and generate a  
    // null reference exception.
```

```
// You cannot modify a property (or use a method) of an
// object that doesn't exist!
conTwo.ConnectionString = "...";
...
}
```

When an error occurs in your code, .NET checks to see whether any *error handlers* appear in the current scope. If the error occurs inside a method, .NET searches for local error handlers and then checks for any active error handlers in the calling code. If no error handlers are found, the page processing is aborted. If you're running the page in Visual Studio, you'll enter debug mode at this point (as you learned in Chapter 2). If you're running the page outside Visual Studio, ASP.NET will either show an error page you've specified or a built-in error message, depending on the settings you've set.

If you enter debug mode because of an error, you can click the Start button again (or press F5) to keep going. Because your code has been derailed by the unexpected, unhandled error, ASP.NET shows a detailed error page that explains the problem (Figure 7-1). These error pages are a development convenience—after you deploy your application, they are replaced by more-general error pages, which you can configure in the IIS web server software (as described in Chapter 26).

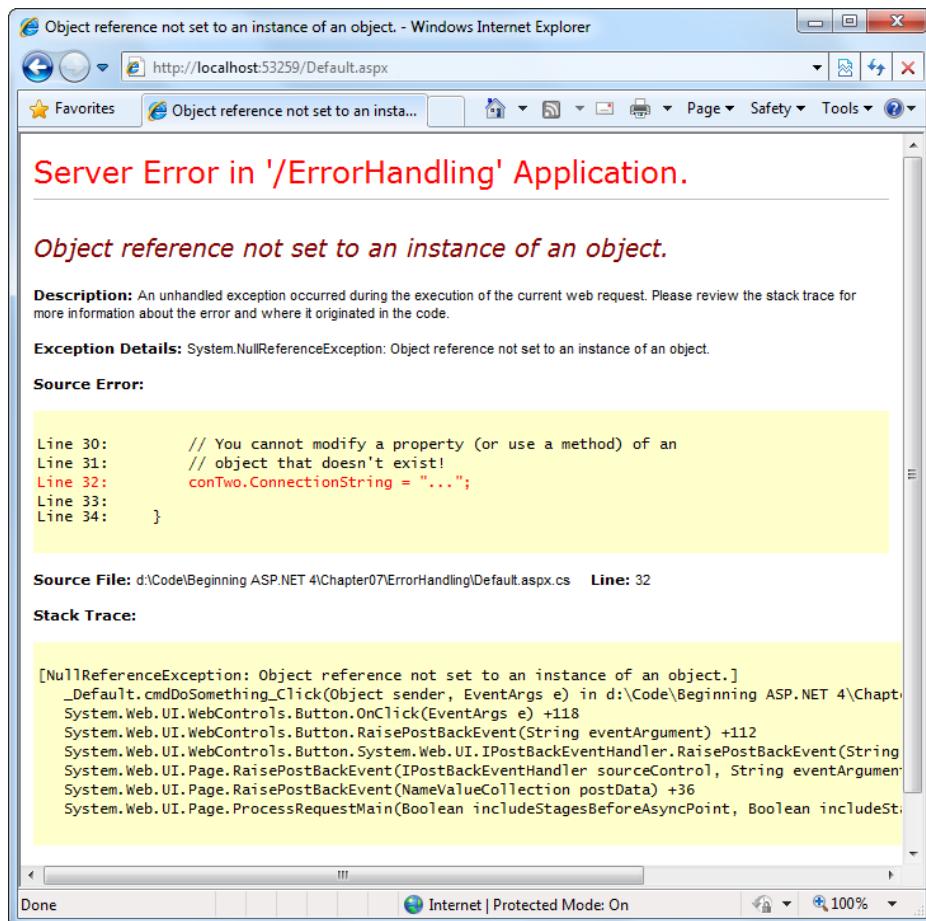


Figure 7-1. A sample error page

Even if an error is the result of invalid input or the failure of a third-party component, an error page can shatter the professional appearance of any application. The application users end up feeling that the application is unstable, insecure, or of poor quality—and they're at least partially correct.

If an ASP.NET application is carefully designed and constructed, an unexpected error page will almost never appear. Errors may still occur because of unforeseen circumstances, but they will be caught in the code and identified. If the error is a critical one that the application cannot solve on its own, it will report a more useful (and user-friendly) page of information that might include a link to a support e-mail or a phone number where the customer can receive additional assistance. You'll look at those techniques in this chapter.

Understanding Exception Handling

Most .NET languages support *structured exception handling*. Essentially, when an error occurs in your application, the .NET Framework creates an exception object that represents the problem. You can catch this object by using an exception handler. If you fail to use an exception handler, your code will be aborted, and the user will see an error page. If you catch the exception, you can notify the user, attempt to resolve the problem, or simply ignore the issue and allow your web page code to keep running.

Structured exception handling provides several key features:

Exceptions are object-based: Each exception provides a significant amount of diagnostic information wrapped into a neat object, instead of a simple message and error code. These exception objects also support an `InnerException` property that allows you to wrap a generic error over the more specific error that caused it. You can even create and throw your own exception objects.

Exceptions are caught based on their type: This allows you to streamline error-handling code without needing to sift through obscure error codes.

Exception handlers use a modern block structure: This makes it easy to activate and deactivate different error handlers for different sections of code and handle their errors individually.

Exception handlers are multilayered: You can easily layer exception handlers on top of other exception handlers, some of which may check for only a specialized set of errors. As you'll see, this gives you the flexibility to handle different types of problems in different parts of your code, thereby keeping your code clean and organized.

Exceptions are a generic part of the .NET Framework: This means they're completely cross-language compatible. Thus, a .NET component written in C# can throw an exception that you can catch in a web page written in VB.

Note Exception handlers are a key programming technique. They allow you to react to problems that runtime because of factors outside your control. However, you obviously shouldn't use exception handlers that might crop up in your code! instead, you need to track down these programmer mistakes at design time and correct them. Visual Studio's debugging features (which were described in Chapter 4) can help with this task.

The Exception Class

Every exception class derives from the base class `System.Exception`. The .NET Framework is full of predefined exception classes, such as `NullReferenceException`, `IOException`, `SQLException`, and so on. The `Exception` class includes the essential functionality for identifying any type of error. Table 7-1 lists its most important members.

Table 7-1. *Exception Properties*

Member	Description
<code>HelpLink</code>	A link to a help document, which can be a relative or fully qualified uniform resource locator (URL) or uniform resource name (URN), such as <code>file:///C:/ACME/MyApp/help.html#Err42</code> . The .NET Framework doesn't use this property, but you can set it in your custom exceptions if you want to use it in your web page code.
<code>InnerException</code>	A nested exception. For example, a method might catch a simple file input/output (IO) error and create a higher-level "operation failed" error. The details about the original error could be retained in the <code>InnerException</code> property of the higher-level error.
<code>Message</code>	A text description with a significant amount of information describing the problem.
<code>Source</code>	The name of the application or object where the exception was raised.
<code>StackTrace</code>	A string that contains a list of all the current method calls on the stack, in the order of most recent to least recent. This is useful for determining where the problem occurred.
<code>TargetSite</code>	A reflection object (an instance of the <code>System.Reflection.MethodBase</code> class) that provides some information about the method where the error occurred. This information includes generic method details such as the method name and the data types for its parameter and return values. It doesn't contain any information about the actual parameter values that were used when the problem occurred.
<code>GetBaseException()</code>	A method useful for nested exceptions that may have more than one layer. It retrieves the original (deepest nested) exception by moving to the base of the <code>InnerException</code> chain.

When you catch an exception in an ASP.NET page, it won't be an instance of the generic `System.Exception` class. Instead, it will be an object that represents a specific type of error. This object will be based on one of the many classes that inherit from `System.Exception`. These include diverse classes such as `DivideByZeroException`, `ArithmetiException`, `IOException`, `SecurityException`, and many more. Some of these classes provide additional details about the error in additional properties.

Visual Studio provides a useful tool to browse through the exceptions in the .NET class library. Simply choose `Debug ➤ Exceptions` from the menu (you'll need to have a project open in order for this to work). The Exceptions dialog box will appear. Expand the Common Language Runtime Exceptions group, which shows a hierarchical tree of .NET exceptions arranged by namespace (see Figure 7-2).

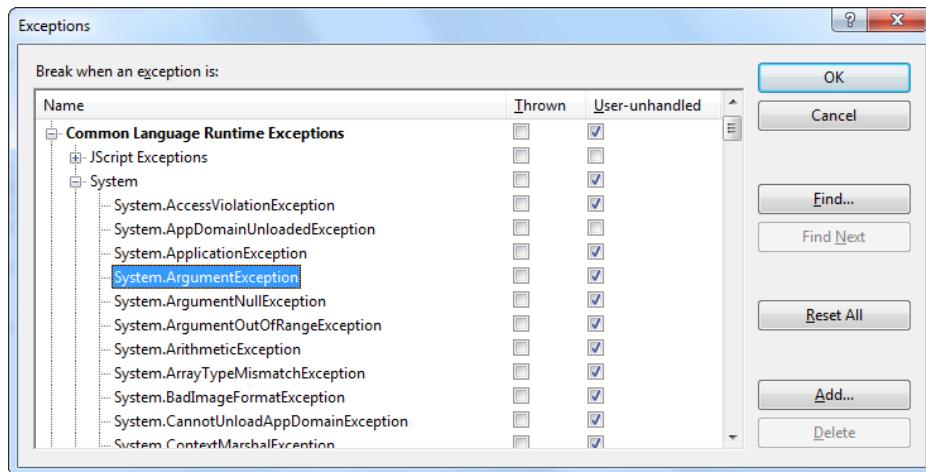


Figure 7-2. Visual Studio’s exception viewer

The Exceptions dialog box allows you to specify which exceptions should be handled by your code when debugging and which exceptions will cause Visual Studio to enter break mode immediately. That means you don’t need to disable your error-handling code to troubleshoot a problem. For example, you could choose to allow your program to handle a common `FileNotFoundException` (which could be caused by an invalid user selection) but instruct Visual Studio to pause execution if an unexpected `DivideByZero` exception occurs.

To set this up, add a check mark in the Thrown column next to the entry for the `System.DivideByZero` exception. This way, you’ll be alerted as soon as the problem occurs. If you don’t add a check mark to the Thrown column, your code will continue, run any exception handlers it has defined, and try to deal with the problem. You’ll be notified only if an error occurs and no suitable exception handler is available.

The Exception Chain

Figure 7-3 shows how the `InnerException` property works. In the specific scenario shown here, a `FileNotFoundException` led to a `NullReferenceException`, which led to a custom `UpdateFailedException`.

Using an exception-handling block, the application can catch the `UpdateFailedException`. It can then get more information about the source of the problem by following the `InnerException` property to the `NullReferenceException`, which in turn references the original `FileNotFoundException`.

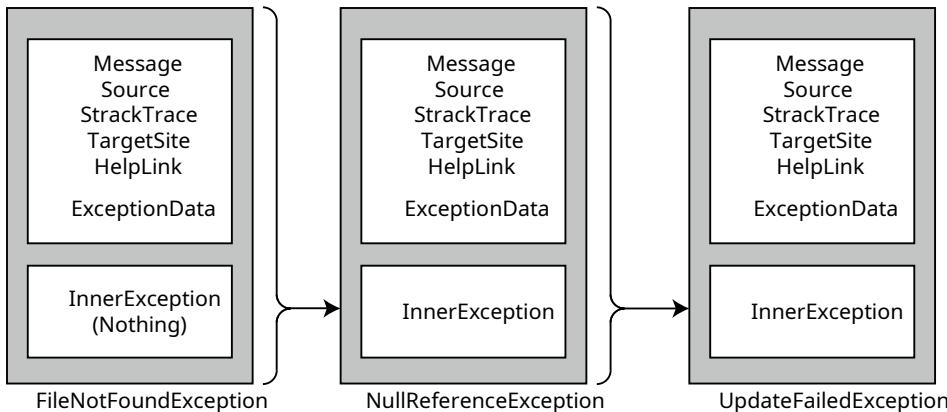


Figure 7-3. Exceptions can be chained together

The `InnerException` property is an extremely useful tool for component-based programming. Generally, it's not much help if a component reports a low-level problem such as a null reference or a divide-by-zero error. Instead, it needs to communicate a more detailed message about which operation failed and what input may have been invalid. The calling code can then often correct the problem and retry the operation.

On the other hand, sometimes you're debugging a bug that lurks deep inside the component itself. In this case, you need to know precisely what caused the error—you don't want to replace it with a higher-level exception that could obscure the root problem. Using an exception chain handles both these scenarios: you receive as many linked exception objects as needed, which can specify information from the least to the most specific error condition.

Handling Exceptions

The first line of defense in an application is to check for potential error conditions before performing an operation. For example, a program can explicitly check whether the divisor is 0 before performing a calculation or whether a file exists before attempting to open it:

```

if (divisor != 0)
{
    // It's safe to divide some number by divisor.
}

if (System.IO.File.Exists("myfile.txt"))
{
    // You can now open the myfile.txt file.
    // However, you should still use exception handling because a variety of
    // problems can intervene (insufficient rights, hardware failure, etc.).
}

```

Even if you perform this basic level of “quality assurance,” your application is still vulnerable. For example, you have no way to protect against all the possible file access problems that occur, including hardware failures or network problems that could arise spontaneously in the middle of an operation. Similarly, you have no way to validate a user ID and password for a database before attempting to open a connection—and even if you did, that technique would be subject to its own set of potential errors. In some cases, it may not be practical to perform the

full range of defensive checks, because they may impose a noticeable performance drag on your application. For all these reasons, you need a way to detect and deal with errors when they occur.

The solution is *structured exception handling*. To use structured exception handling, you wrap potentially problematic code in the special block structure shown here:

```
try
{
    // Risky code goes here (opening a file, connecting to a database, and so on).
}
catch
{
    // An error has been detected. You can deal with it here.
}
finally
{
    // Time to clean up, regardless of whether or not there was an error.
}
```

The try statement enables error handling. Any exceptions that occur in the following lines can be “caught” automatically. The code in the catch block will be executed when an error is detected. And either way, whether a bug occurs or not, the finally block of the code will be executed last. This allows you to perform some basic cleanup, such as closing a database connection. The finally code is important because it will execute even if an error has occurred that will prevent the program from continuing. In other words, if an unrecoverable exception halts your application, you’ll still have the chance to release resources.

The act of catching an exception neutralizes it. If all you want to do is render a specific error harmless, you don’t even need to add any code in the catch block of your error handler. Usually, however, this portion of the code will be used to report the error to the user or log it for future reference. In a separate component (such as a business object), this code might handle the exception, perform some cleanup, and then rethrow it to the calling code, which will be in the best position to remedy it or alert the user. Or it might actually create a new exception object with additional information and throw that.

Catching Specific Exceptions

Structured exception handling is particularly flexible because it allows you to catch specific types of exceptions. To do so, you add multiple catch statements, each one identifying the type of exception (and providing a new variable to catch it in), as follows:

```
try
{
    // Risky database code goes here.
}
catch (System.Data.SqlClient.SqlException err)
{
    // Catches common problems like connection errors.
}
catch (System.NullReferenceException err)
{
    // Catches problems resulting from an uninitialized object.
}
```

An exception will be caught as long as it's an instance of the indicated class or if it's derived from that class. In other words, if you use this statement:

```
catch (Exception err)
```

you will catch any exception, because every exception object is derived from the System.Exception base class.

Exception blocks work a little like conditional code. As soon as a matching exception handler is found, the appropriate catch code is invoked. Therefore, you must organize your catch statements from most specific to least specific:

```
try
{
    // Risky database code goes here.
}
catch (System.Data.SqlClient.SqlException err)
{
    // Catches common problems like connection errors.
}
catch (System.NullReferenceException err)
{
    // Catches problems resulting from an uninitialized object.
}
catch (System.Exception err)
{
    // Catches any other errors.
}
```

Ending with a catch statement for the base Exception class is often a good idea to make sure no errors slip through. However, in component-based programming, you should make sure you intercept only those exceptions you can deal with or recover from. Otherwise, it's better to let the calling code catch the original error.

Determining the exCeptions you neeD to CatCh

When you're using classes from the .nET Framework, you may not know what exceptions you need to handle. Fortunately, the Visual Studio Help can fill you in.

The trick is to look up the method or constructor you're using in the class library reference. One fast way to jump to a specific method is to use the Help index—just type in the class name, followed by a period, followed by the method name, as in File.open (which is a method you'll use to open files in Chapter 17). If there is more than one overloaded version of the method, you'll see a page that lists them all, and you'll need to click the one that has the parameters you want.

After you find the right method, scroll through the method documentation until you find a section named Exceptions. This section lists all the possible exceptions that this method can throw. For example, if you look up the File.open() method, you'll find that possible exceptions include directoryNotFoundException, FileNotFoundException, UnauthorizedAccessException, and so on. You probably won't write a try block for each possible exception. However, you should still know about all of them so you can decide which exceptions you want to handle separately.

Using Nested Exception Handlers

When an exception is thrown, .NET tries to find a matching catch statement in the current method. If the code isn't in a local structured exception block or if none of the catch statements matches the exception, .NET will move up the call stack one level at a time, searching for active exception handlers.

Consider the example shown here, where the Page.Load event handler calls a private DivideNumbers() method:

```
protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        DivideNumbers(5, 0);
    }
    catch (DivideByZeroException err)
    {
        // Report error here.
    }
}

private decimal DivideNumbers(decimal number, decimal divisor)
{
    return number/divisor;
}
```

In this example, the DivideNumbers() method lacks any sort of exception handler. However, the DivideNumbers() method call is made inside a try block, which means the problem will be caught further upstream in the calling code. This is a good approach because the DivideNumbers() routine could be used in a variety of circumstances (or if it's part of a component, in various types of applications). It really has no access to any kind of user interface and can't directly report an error. Only the calling code is in a position to determine whether the problem is a serious one or a minor one, and only the calling code can prompt the user for more information or report error details in the web page.

Note in this example, great care is taken to use the decimal data type rather than the more common double data type. That's because contrary to what you might expect, it *is* acceptable to divide a double by 0. The result is the special value double.Positiveinfinity (or double.Negativeinfinity if you divide a negative number by 0).

You can also overlap exception handlers in such a way that different exception handlers filter out different types of problems. Here's one such example:

```
protected void Page_Load(Object sender, EventArgs e)
{
    try
    {
        decimal average = GetAverageCost(DateTime.Now);
    }
    catch (DivideByZeroException err)
    {
        // Report error here.
    }
}
```

```
private decimal GetAverageCost(DateTime saleDate)
{
    try
    {
        // Use Database access code here to retrieve all the sale records
        // for this date, and calculate the average.
    }
    catch (System.Data.SqlClient.SqlException err)
    {
        // Handle a database-related problem.
    }
    finally
    {
        // Close the database connection.
    }
}
```

Dissecting the Code . . .

You should be aware of the following points:

- If an `SqlException` occurs during the database operation, it will be caught in the `GetAverageCost()` method.
- If a `DivideByZeroException` occurs (for example, the method receives no records but still attempts to calculate an average), the exception will be caught in the calling `Page.Load` event handler.
- If another problem occurs (such as a null reference exception), no active exception handler exists to catch it. In this case, .NET will search through the entire call stack without finding a matching catch statement in an active exception handler and will generate a runtime error, end the program, and return a page with exception information.

Exception Handling in Action

You can use a simple program to test exceptions and see what sort of information is retrieved. This program allows a user to enter two values and attempts to divide them. It then reports all the related exception information in the page (see Figure 7-4).

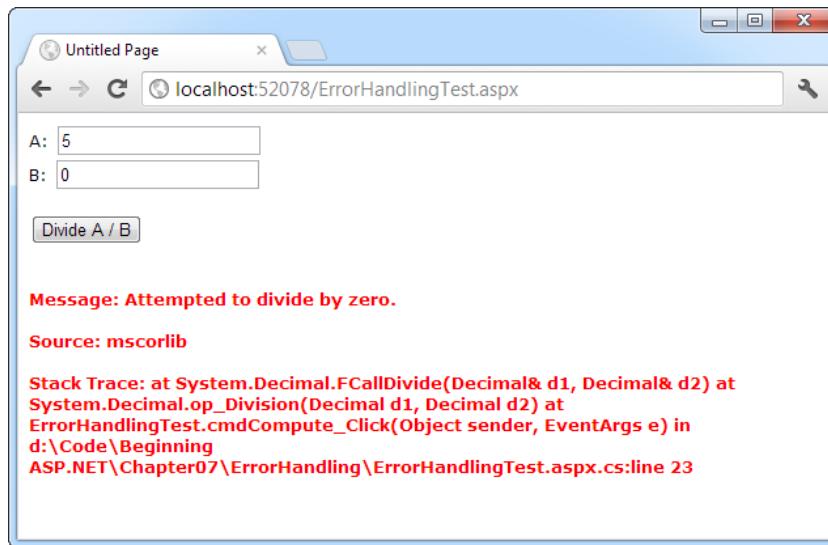


Figure 7-4. Catching and displaying exception information

Obviously, you can easily prevent this exception from occurring by using extra code-safety checks, or you can elegantly resolve it by using the validation controls. However, this code provides a good example of how you can deal with the properties of an exception object. It also gives you a good idea about the sort of information that will be returned.

Here's the page class code for this example:

```

public partial class ErrorHandlingTest : System.Web.UI.Page
{
    protected void cmdCompute_Click(Object sender, EventArgs e)
    {
        try
        {
            decimal a, b, result;
            a = Decimal.Parse(txtA.Text);
            b = Decimal.Parse(txtB.Text);
            result = a / b;
            lblResult.Text = result.ToString();
            lblResult.ForeColor = System.Drawing.Color.Black;
        }
        catch (Exception err)
        {
            lblResult.Text = "<b>Message:</b> " + err.Message;
            lblResult.Text += "<br /><br />";
            lblResult.Text += "<b>Source:</b> " + err.Source;
            lblResult.Text += "<br /><br />";
            lblResult.Text += "<b>Stack Trace:</b> " + err.StackTrace;
            lblResult.ForeColor = System.Drawing.Color.Red;
        }
    }
}

```

Note that as soon as the error occurs, execution is transferred to an exception handler. The code in the try block isn't completed. It's for that reason that the result for the label is set in the try block. These lines will be executed only if the division code runs error-free.

You'll see many more examples of exception handling throughout this book. The data access chapters in Part 4 of this book show the best practices for exception handling when accessing a database.

Mastering Exceptions

Keep in mind these points when working with structured exception handling:

Break down your code into multiple try/catch blocks: If you put all your code into one exception handler, you'll have trouble determining where the problem occurred. You have no way to "resume" the code in a try block. This means that if an error occurs at the beginning of a lengthy try block, you'll skip a large amount of code. The rule of thumb is to use one exception handler for one related task (such as opening a file and retrieving information).

Report all errors: During debugging, portions of your application's error-handling code may mask easily correctable mistakes in your application. To prevent this from happening, make sure you report all errors, and consider leaving out some error-handling logic in early builds.

Don't use exception handlers for every statement: Simple code statements (assigning a constant value to a variable, interacting with a control, and so on) may cause errors during development testing but will not cause any future problems once perfected. Error handling should be used when you're accessing an outside resource or dealing with supplied data that you have no control over (and thus may be invalid).

Throwing Your Own Exceptions

You can also define your own exception objects to represent custom error conditions. All you need to do is create an instance of the appropriate exception class and then use the throw statement.

The next example introduces a modified DivideNumbers() method. It explicitly checks whether the specified divisor is 0 and then manually creates and throws an instance of the DivideByZeroException class to indicate the problem, rather than attempt the operation. Depending on the code, this pattern can save time by eliminating some unnecessary steps, or it can prevent a task from being initiated if it can't be completed successfully.

```
protected void Page_Load(Object sender, EventArgs e)
{
    try
    {
        DivideNumbers(5, 0);
    }
    catch (DivideByZeroException err)
    {
        // Report error here.
    }
}
```

```
private decimal DivideNumbers(decimal number, decimal divisor)
{
    if (divisor == 0)
    {
        DivideByZeroException err = new DivideByZeroException();
        throw err;
    }
    else
    {
        return number/divisor;
    }
}
```

Alternatively, you can create a .NET exception object and specify a custom error message by using a different constructor:

```
private decimal DivideNumbers(decimal number, decimal divisor)
{
    if (divisor == 0)
    {
        DivideByZeroException err = new DivideByZeroException(
            "You supplied 0 for the divisor parameter. You must be stopped.");
        throw err;
    }
    else
    {
        return number/divisor;
    }
}
```

In this case, any ordinary exception handler will still catch the DivideByZeroException. The only difference is that the error object has a modified Message property that contains the custom string. Figure 7-5 shows the resulting exception.

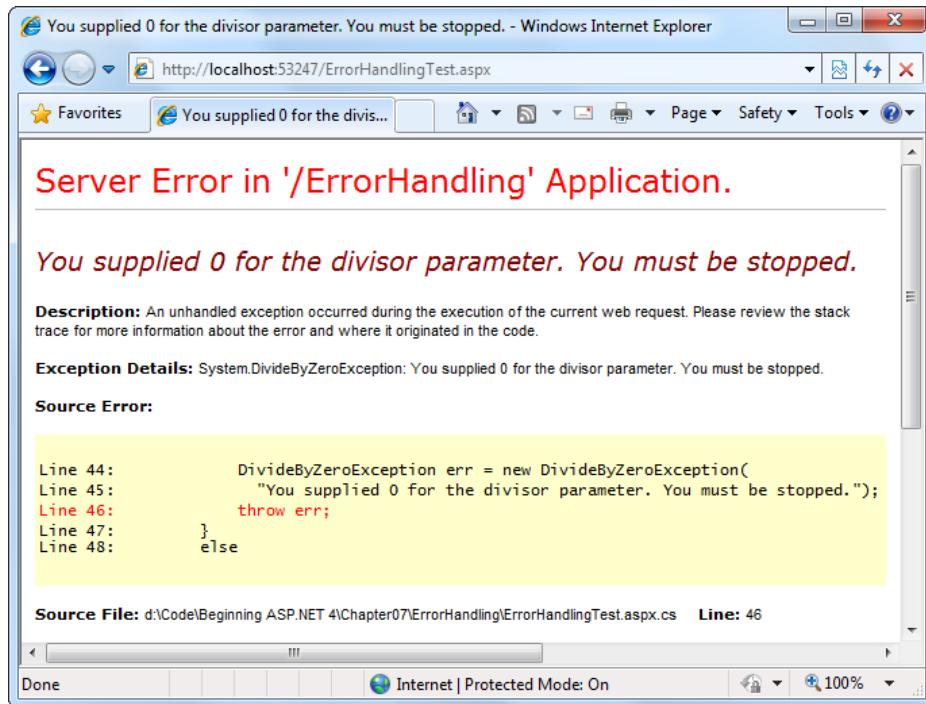


Figure 7-5. Standard exception, custom message

Throwing an exception is most useful in component-based programming. In component-based programming, your ASP.NET page is creating objects and calling methods from a class defined in a separately compiled assembly. In this case, the class in the component needs to be able to notify the calling code (the web application) of any errors. The component should handle recoverable errors quietly and not pass them up to the calling code. On the other hand, if an unrecoverable error occurs, it should always be indicated with an exception and never through another mechanism (such as a return code). For more information about component-based programming, refer to Chapter 22.

If you can find an exception in the class library that accurately reflects the problem that has occurred, you should throw it. If you need to return additional or specialized information, you can create your own custom exception class.

Custom exception classes should always inherit from `System.ApplicationException`, which itself derives from the base `Exception` class. This allows .NET to distinguish between two broad classes of exceptions—those you create and those that are native to the .NET Framework.

When you create an exception class, you can add properties to record additional information. For example, here is a special class that records information about the failed attempt to divide by zero:

```
public class CustomDivideByZeroException : ApplicationException
{
    // Add a variable to specify the "other" number.
    // This might help diagnose the problem.
    public decimal DividingNumber;
}
```

As you'll see, after you create a custom exception, you have the opportunity to stuff it with extra information. Of course, it's unlikely that your code would use a custom exception class for something as basic as dividing by

zero—instead, the code would be more likely to have an exception related to some application-specific task, such as a `ValidatingPromoCodeException` or an `InvalidOrderRequestException`. Still, the steps you use to create the exception remain the same.

Note You can define the custom exception class directly in your `.aspx.cs` code file. However, a more common approach is to use a separate code file. To do this, right-click your application in the Solution Explorer and choose Add new item. Then pick Class from the list of templates, enter a file name, and click Add. Visual Studio will create a new code file in the `App_Code` subfolder of your website. (it's up to you whether you create a separate code file for every class or put several classes in the same code file. Both approaches have the same effect.)

You can throw this custom exception like this:

```
private decimal DivideNumbers(decimal number, decimal divisor)
{
    if (divisor == 0)
    {
        CustomDivideByZeroException err = new CustomDivideByZeroException();
        err.DividingNumber = number;
        throw err;
    }
    else
    {
        return number/divisor;
    }
}
```

To perfect the custom exception, you need to supply it with the three standard constructors. This allows your exception class to be created in the standard ways that every exception supports:

- On its own, with no arguments
- With a custom message
- With a custom message and an exception object to use as the inner exception

These constructors don't actually need to contain any code. All these constructors need to do is forward the parameters to the base class (the constructors in the inherited `ApplicationException` class) by using the `base` keyword, as shown here:

```
public class CustomDivideByZeroException : ApplicationException
{
    // Add a variable to specify the "other" number.
    private decimal dividingNumber;
    public decimal DividingNumber
    {
        get { return dividingNumber; }
        set { dividingNumber = value; }
    }

    public CustomDivideByZeroException() : base()
    {}
```

```

public CustomDivideByZeroException(string message) : base(message)
{
}

public CustomDivideByZeroException(string message, Exception inner) :
    base(message, inner)
{
}
}
```

The third constructor is particularly useful for component programming. It allows you to set the InnerException property with the exception object that caused the original problem. The next example shows how you could use this constructor with a component class called ArithmeticUtility:

```

public class ArithmeticUtilityException : ApplicationException
{
    public ArithmeticUtilityException() : base()
    {}

    public ArithmeticUtilityException(string message) : base(message)
    {}

    public ArithmeticUtilityException(string message, Exception inner) :
        base(message, inner)
    {}
}

public class ArithmeticUtility
{
    private decimal Divide(decimal number, decimal divisor)
    {
        try
        {
            return number/divisor;
        }
        catch (Exception err)
        {
            // Create an instance of the specialized exception class,
            // and place the original exception object (for example, a
            // DivideByZeroException)in the InnerException property.
            ArithmeticUtilityException errNew =
                new ArithmeticUtilityException("Calculation error", err);

            // Now throw the new exception.
            throw errNew;
        }
    }
}
```

Remember, custom exception classes are really just a standardized way for one class to communicate an error to a different portion of code. If you haven't reached that level of design—in other words, you aren't yet putting useful bits of your code in a separate class, outside the web page, or in a completely separate component (see Chapter 22)—you probably don't need to create custom exception classes.

Using Page Tracing

Visual Studio's debugging tools and ASP.NET's detailed error pages are extremely helpful when you're testing an application. However, sometimes you need a way to identify problems after you've deployed an application, when you don't have Visual Studio to rely on.

You could try to identify these errors by recording diagnostic information in an event log, but this assumes that someone will actually review the log regularly. More aggressively, you could display some information directly in the web page. The problem with this strategy is that you need to remove (or at least comment out) all this extra code before you deploy your web application. Otherwise, your website users could see strange debugging messages when they least expect it.

Fortunately, there's an easier way to solve the problem without resorting to a homegrown solution. ASP.NET provides a feature called *tracing* that gives you a far more convenient and flexible way to report diagnostic information.

Enabling Tracing

To use tracing, you need to explicitly enable it. There are several ways to switch on tracing. One of the easiest ways is by adding an attribute to the Page directive in the .aspx file:

```
<%@ Page Trace="true" ... %>
```

You can also enable tracing by using the built-in `Trace` object (which is an instance of the `System.Web.TraceContext` class). Here's an example of how you might turn on tracing in the `Page.Load` event handler:

```
protected void Page_Load(Object sender, EventArgs e)
{
    Trace.IsEnabled = true;
}
```

This technique is useful because it allows you to enable or disable tracing for a page under specific circumstances that you test for in your code.

Note that by default, after you enable tracing, it will apply only to local requests. (In other words, if you're working with a deployed web application, you need to make your requests from the web browser on the web server computer.) This limitation prevents end users from seeing the tracing information. If you need to trace a web page from an offsite location, you'll need to enable remote tracing by changing some settings in the `web.config` file. (You can find information about modifying these settings in the "Performing Application-Level Tracing" section later in this chapter.) Once you've enabled remote tracing, you can use your code to selectively turn on tracing—for example, for specific users.

Tracing Information

ASP.NET tracing automatically provides a lengthy set of standard, formatted information. This information allows you to monitor several important aspects of your application, such as the contents of the current session and the time taken to execute portions of code.

Figure 7-6 shows what this information looks like. To build this example, you can start with any basic ASP.NET page. Shown here is a rudimentary ASP.NET page with just a label and a button.

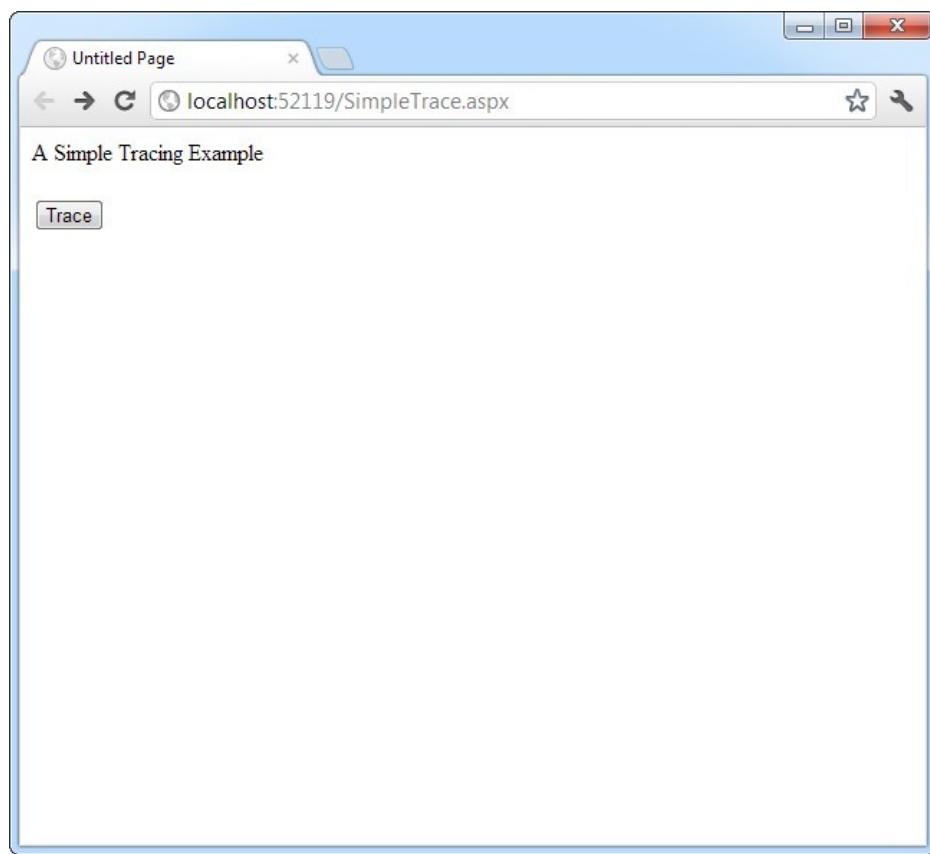


Figure 7-6. A simple ASP.NET page

On its own, this page does very little, displaying a single line of text. However, if you click the button, tracing is enabled by setting the `Trace.IsEnabled` property to true (as shown in the previous code snippet). When the page is rendered, it will include a significant amount of diagnostic information, as shown in Figure 7-7.

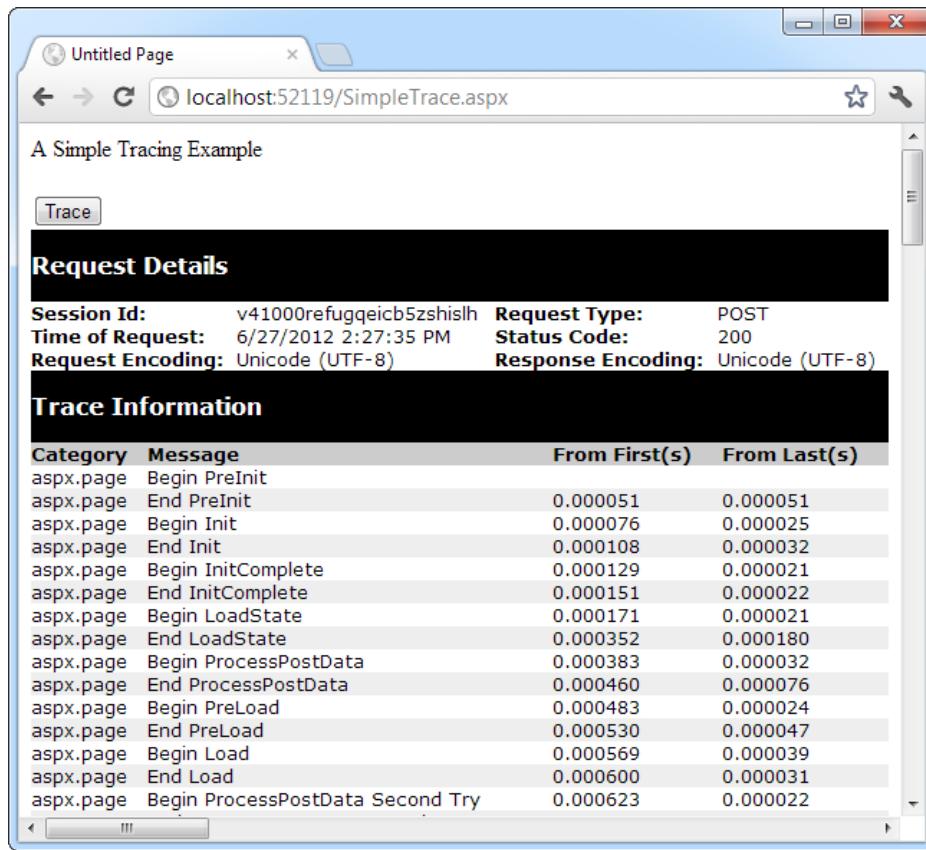


Figure 7-7. Tracing the simple ASP.NET page

Tracing information is provided in several categories, which are described in the following sections. Depending on your page, you may not see all the tracing information. For example, if the page request doesn't supply any query string parameters, you won't see the `QueryString` collection. Similarly, if no data is being held in application or session state, you won't see those sections either.

Tip if you're using style sheets, your rules may affect the formatting and layout of the trace information, making it difficult to read. if this becomes a problem, you can use application-level tracing, as described in this chapter (see the "Performing Application-level Tracing" section).

Request Details

This section includes some basic information such as the current session ID, the time the web request was made, and the type of web request and encoding (see Figure 7-8). Most of these details are fairly uninteresting, and you won't spend much time looking at them. The exception is the session ID. By watching for when the session ID changes, you'll know when a new session is created. (Sessions are used to store information for a specific user in between page requests. You'll learn about them in Chapter 8.)

Request Details			
Session Id:	v41000refugqei5zshishl	Request Type:	POST
Time of Request:	6/27/2012 2:18:38 PM	Status Code:	200
Request Encoding:	Unicode (UTF-8)	Response Encoding:	Unicode (UTF-8)

Figure 7-8. Request details

Trace Information

Trace information shows the stages of processing that the page went through before being sent to the client (see Figure 7-9). Each section has additional information about how long the processing took to complete, as a measure from the start of the first stage (From First) and as a measure from the start of the previous stage (From Last). If you add your own trace messages (a technique described shortly), they will also appear in this section.

Trace Information			
Category	Message	From First(s)	From Last(s)
aspx.page	End RaisePostBackEvent		
aspx.page	Begin LoadComplete	4.83301648673225E-05	0.000048
aspx.page	End LoadComplete	8.85587414042846E-05	0.000040
aspx.page	Begin PreRender	0.000119847634266366	0.000031
aspx.page	End PreRender	0.000163987322411089	0.000044
aspx.page	Begin PreRenderComplete	0.00019695240596221	0.000033
aspx.page	End PreRenderComplete	0.000227682568594612	0.000031
aspx.page	Begin SaveState	0.000686400087161916	0.000459
aspx.page	End SaveState	0.000843403281702004	0.000157
aspx.page	Begin SaveStateComplete	0.000880279476860886	0.000037
aspx.page	End SaveStateComplete	0.000917155672019768	0.000037
aspx.page	Begin Render	0.00094760646953733	0.000030
aspx.page	End Render	0.00136246366507475	0.000415

Figure 7-9. Trace information

Control Tree

The control tree shows you all the controls on the page, indented to show their hierarchy (which controls are contained inside other controls), as shown in Figure 7-10. In this simple page example, the control tree includes a button named cmdTrace, which is explicitly defined in the web page markup. ASP.NET also adds literal controls automatically to represent spacing and any other static elements that aren't server controls (such as text or ordinary HTML tags). These controls appear between the buttons in this example, and have automatically generated names such as ctl00, ctl01, ctl02, and so on.

One useful feature of this section is the ViewState Size column, which tells you the number of bytes of space required to persist the current information in the control. This can help you gauge whether enabling control state is detracting from performance, particularly when working with data-bound controls such as the GridView.

Control Tree				
Control UniqueID	Type	Render Size Bytes (including children)	ViewState Size Bytes (excluding children)	ControlState Size Bytes (excluding children)
Page	ASP.simpletrace_aspx	840	0	0
ctl02	System.Web.UI.LiteralControl	151	0	0
ctl00	System.Web.UI.HtmlControls.HtmlHead46	0	0	0
ctl01	System.Web.UI.HtmlControls.HtmlTitle33	0	0	0
ctl03	System.Web.UI.LiteralControl	14	0	0
form1	System.Web.UI.HtmlControls.HtmlForm609	0	0	0
ctl04	System.Web.UI.LiteralControl	77	0	0
cmdTrace	System.Web.UI.WebControls.Button	67	0	0
ctl05	System.Web.UI.LiteralControl	12	0	0
ctl06	System.Web.UI.LiteralControl	20	0	0

Figure 7-10. Control tree

Session State and Application State

These sections display every item that is in the current session or application state. Each item in the appropriate state collection is listed with its name, type, and value. If you’re storing simple pieces of string information, the value is straightforward—it’s the actual text in the string. If you’re storing an object, .NET calls the object’s `ToString()` method to get an appropriate string representation. For complex objects that don’t override `ToString()` to provide anything useful, the result may be just the class name.

Figure 7-11 shows the session state section after you’ve added two items to session state (an ordinary string and a `DataSet` object). Chapter 8 has more about using session state.

Session State		
Session Key	Type	Value
TestString	System.String	This is just a string.
MyDataSet	System.Data.DataSet	System.Data.DataSet

Figure 7-11. Session state

Request Cookies and Response Cookies

These sections display the cookies that were sent by the web browser with the request for this page and display the cookies that were returned by the web server with the response. ASP.NET shows the content and the size of each cookie in bytes.

Figure 7-12 shows an example with a page that uses a cookie named `Preferences` that stores a single piece of information: a username. (You learned to write the code that creates this cookie in Chapter 8.) In addition, the web browser receives a cookie named `ASP.NET_SessionId`, which ASP.NET creates automatically to store the current session ID.

Request Cookies Collection		
Name	Value	Size
ASP.NET_SessionId	Izydtbz0iw5dvoyhne2oevz3	42
Response Cookies Collection		
Name	Value	Size
Preferences	(Name=Jackson Polenta)	32

Figure 7-12. Cookies collections

There's one quirk with the list of cookies in the trace information. If you haven't created at least one custom cookie of your own, you won't see *any* cookies, including the ones that ASP.NET creates automatically (such as the session cookie). ASP.NET assumes that if you aren't using cookies yourself, you aren't interested in seeing these details.

Headers Collection

This section lists all the HTTP headers (see Figure 7-13). Technically, the headers are bits of information that are sent to the server as part of a request. They include information about the browser making the request, the types of content it supports, and the language it uses. In addition, the Response Headers Collection lists the headers that are sent to the client as part of a response (just before the actual HTML that's shown in the browser). The set of response headers is smaller, and it includes details such as the version of ASP.NET and the type of content that's being sent (text/html for web pages).

Generally, you don't need to use the header information directly. Instead, ASP.NET takes this information into account automatically.

Headers Collection	
Name	Value
Cache-Control	max-age=0
Connection	keep-alive
Content-Length	214
Content-Type	application/x-www-form-urlencoded
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Charset	ISO-8859-1,utf-8;q=0.7,*;q=0.3
Accept-Encoding	gzip,deflate,sdch
Accept-Language	en-GB,en-US;q=0.8,en;q=0.6
Cookie	ASP.NET_SessionId=v41000refugqeicb5zshishlh;.ASPXAUTH=4D15116CAAFE32AE3F1E0517C7F5C
Host	localhost:52078
Referer	http://localhost:52078/ErrorHandling/SimpleTrace.aspx
User-Agent	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.108
Origin	http://localhost:52078

Response Headers Collection	
Name	Value
X-AspNet-Version	4.0.30319
Cache-Control	private
Content-Type	text/html

Figure 7-13. Headers collection

Form Collection

This section lists the posted-back form information. The form information includes all the values that are submitted by web controls, such as the text in a text box and the current selection in a list box. The ASP.NET web controls pull the information they need out of the form collection automatically, so you rarely need to worry about it.

Figure 7-14 shows the form values for the simple page shown in Figure 7-6. It includes the hidden view state field, another hidden field that's used for event validation (a low-level ASP.NET feature that helps prevent people

from tampering with your web pages before posting them back), and a field for the cmdTrace button, which is the only web control on the page.

Form Collection	
Name	Value
__VIEWSTATE	/wEPDwUKMTQ2OTkzNDMyMWRk6PdTEsrgNkgFP9+LyoOOXsjohnM=
cmdTrace	Trace
__EVENTVALIDATION	/wEWAgl9lt3/BQLqhI2yDuBna1E/STtAj+7NCZA4/ezpy4LZ

Figure 7-14. Form collection

Query String Collection

This section lists the variables and values submitted in the query string. You can see this information directly in the web page URL (in the address box in the browser). However, if the query string consists of several different values and contains a lot of information, it may be easier to review the individual items in the trace display.

Figure 7-15 shows the information for a page that was requested with two query string values, one named *search* and the other named *style*. You can try this with the SimpleTrace.aspx page by typing in ?search=cat&style=full at the end of the URL in the address box of your web browser.

Querystring Collection	
Name	Value
search	cat
style	full

Figure 7-15. Query string collection

Server Variables

This section lists all the server variables and their contents. You don't generally need to examine this information. Note also that if you want to examine a server variable programmatically, you can do so by name with the built-in Request.ServerVariables collection or by using one of the more useful higher-level properties from the Request object.

Writing Trace Information

In addition to the standard trace information, you'll often want to generate your own tracing messages. For example, you might want to output the value of a variable at various points in execution so you can compare it with an expected value. Similarly, you might want to output messages when the code reaches certain points in execution so you can verify that various procedures are being used (and are used in the order you expect). Once again, these are tasks you can also achieve using Visual Studio debugging, but tracing is an invaluable technique when you're working with a web application that's been deployed to a web server.

To write a custom trace message, you use the Write() method or the Warn() method of the built-in Trace object. These methods are equivalent. The only difference is that Warn() displays the message in red lettering, which makes it easier to distinguish from other messages in the list. Here's a code snippet that writes a trace message when the user clicks a button:

```
protected void cmdWrite_Click(Object sender, EventArgs e)
{
    Trace.Write("About to place an item in session state.");
    Session["Test"] = "Contents";
    Trace.Write("Placed item in session state.");
}
```

These messages appear in the Trace Information section of the page, along with the default messages that ASP.NET generates automatically (see Figure 7-16).

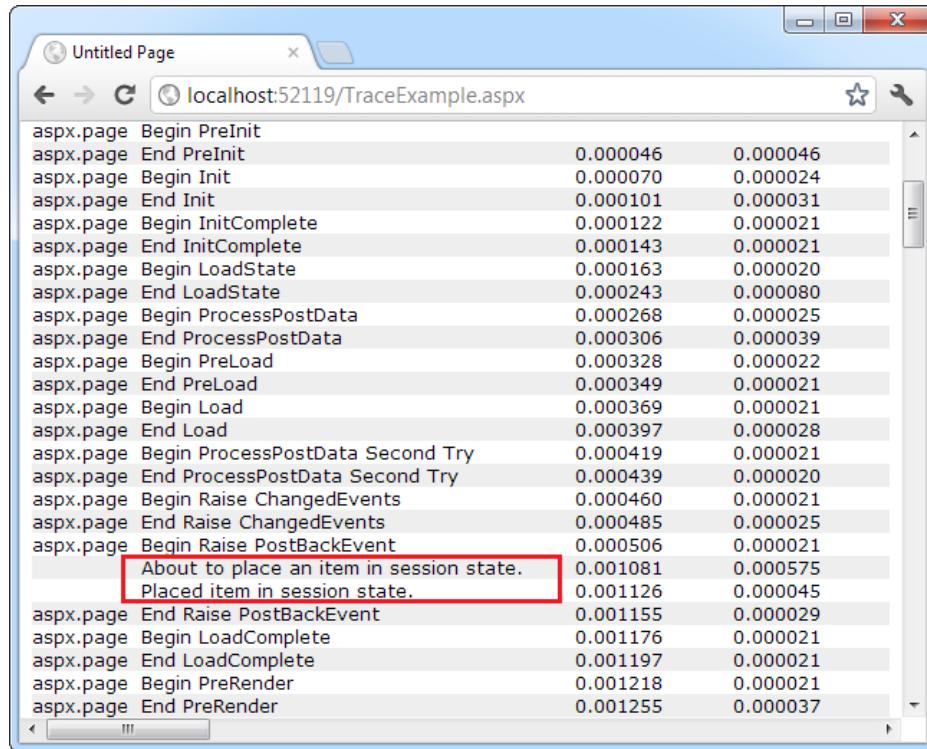
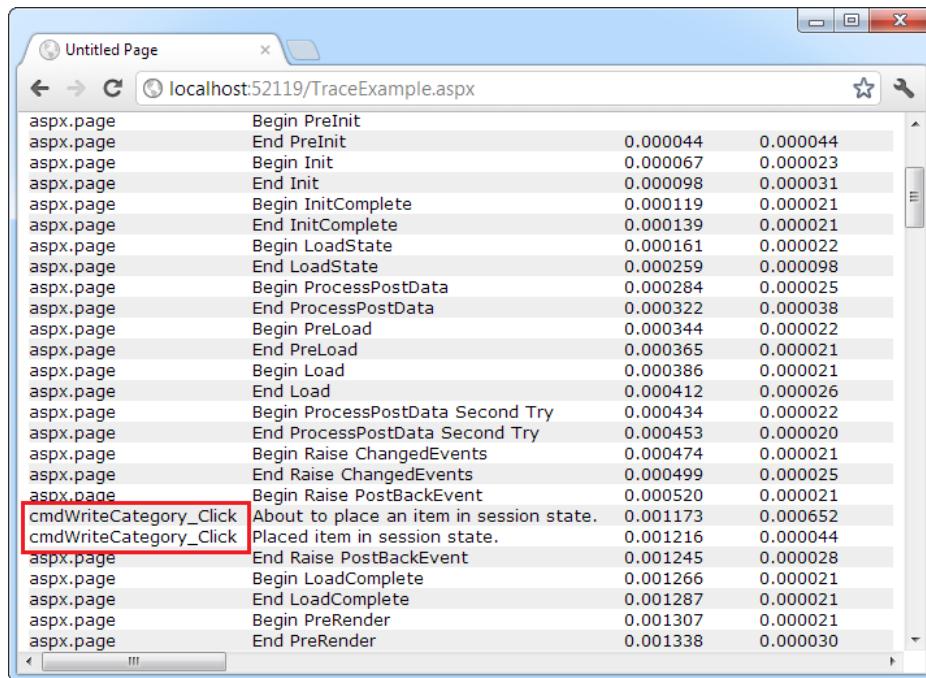


Figure 7-16. Custom trace messages

You can also use an overloaded method of Write() or Warn() that allows you to specify the category. A common use of this field is to indicate the current method, as shown in Figure 7-17.

**Figure 7-17.** A categorized trace message

```
protected void cmdWriteCategory_Click(Object sender, EventArgs e)
{
    Trace.Write("cmdWriteCategory_Click",
    "About to place an item in session state.");
    Session["Test"] = "Contents";
    Trace.Write("cmdWriteCategory_Click",
    "Placed item in session state.");
}
```

Alternatively, you can supply category and message information with an exception object that will automatically be described in the trace log, as shown in Figure 7-18.

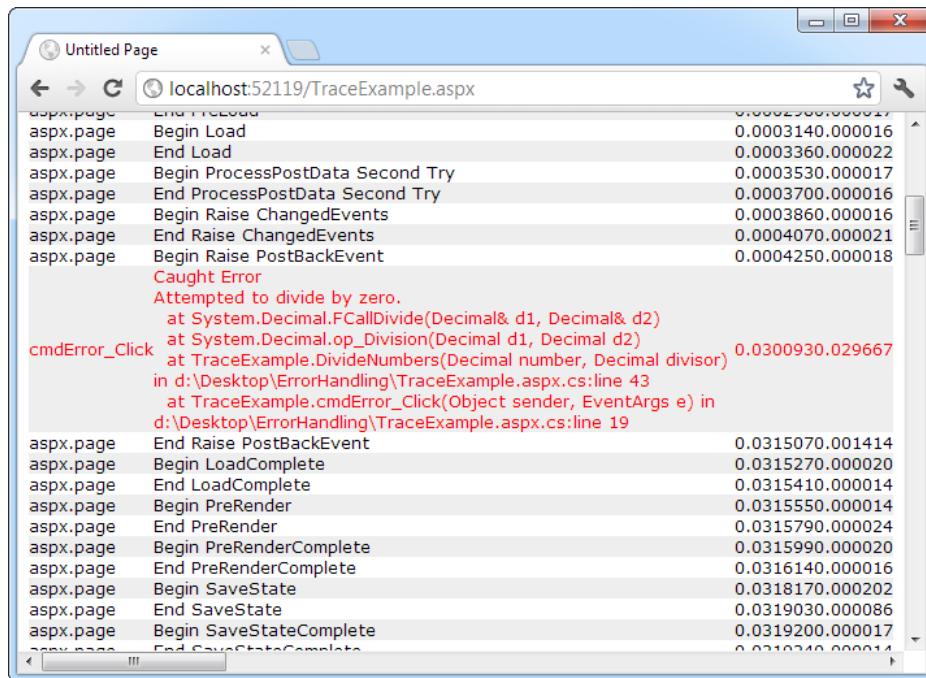


Figure 7-18. An exception trace message

```
protected void cmdError_Click(Object sender, EventArgs e)
{
    try
    {
        DivideNumbers(5, 0);
    }
    catch (Exception err)
    {
        Trace.Warn("cmdError_Click", "Caught Error", err);
    }
}

private decimal DivideNumbers(decimal number, decimal divisor)
{
    return number/divisor;
}
```

By default, trace messages are listed in the order they were written by your code. Alternatively, you can specify that messages should be sorted by category by using the `TraceMode` attribute in the `Page` directive:

```
<%@ Page Trace="true" TraceMode="SortByCategory" %>
```

or the `TraceMode` property of the `Trace` object in your code:

```
Trace.TraceMode = TraceMode.SortByCategory;
```

Performing Application-Level Tracing

Application-level tracing allows you to enable tracing for an entire application. However, the tracing information won't be displayed in the page. Instead, it will be collected and stored in memory for a short amount of time. You can review the recently traced information by requesting a special URL. Application-level tracing provides several advantages. The tracing information won't be mangled by the formatting and layout in your web page, you can compare trace information from different requests, and you can review the trace information that's recorded for someone else's request.

To enable application-level tracing, you need to modify settings in the web.config file, as shown here:

```
<configuration>
  <system.web>
    <trace enabled="true" requestLimit="10" pageOutput="false" />
    ...
  </system.web>
</configuration>
```

Table 7-2 lists the tracing options.

Table 7-2. Tracing Options

Attribute	Values	Description
enabled	true, false	Turns application-level tracing on or off.
requestLimit	Any integer (for example, 10)	Stores tracing information for a maximum number of HTTP requests. Unlike page-level tracing, this allows you to collect a batch of information from multiple requests. When the maximum is reached, ASP.NET may discard the information from the oldest request (which is the default behavior) or the information from the new request, depending on the mostRecent setting.
pageOutput	true, false	Determines whether tracing information will be displayed on the page (as it is with page-level tracing). If you choose false, you'll still be able to view the collected information by requesting trace.axd from the virtual directory where your application is running.
traceMode	SortByTime, SortByCategory	Determines the sort order of trace messages. The default is SortByTime.
localOnly	true, false	Determines whether tracing information will be shown only to local clients (clients using the same computer) or can be shown to remote clients as well. By default, this is true, and remote clients cannot see tracing information.
mostRecent	true, false	Keeps only the most recent trace messages if true. When the requestLimit maximum is reached, the information for the oldest request is abandoned every time a new request is received. If false (the default), ASP.NET stops collecting new trace messages when the limit is reached.

To view tracing information, you request the trace.axd file in the web application's root directory. This file doesn't actually exist; instead, ASP.NET automatically intercepts the request and interprets it as a request for the tracing information. It will then list the most recent collected requests, provided you're making the request from the local machine or have enabled remote tracing (see Figure 7-19).

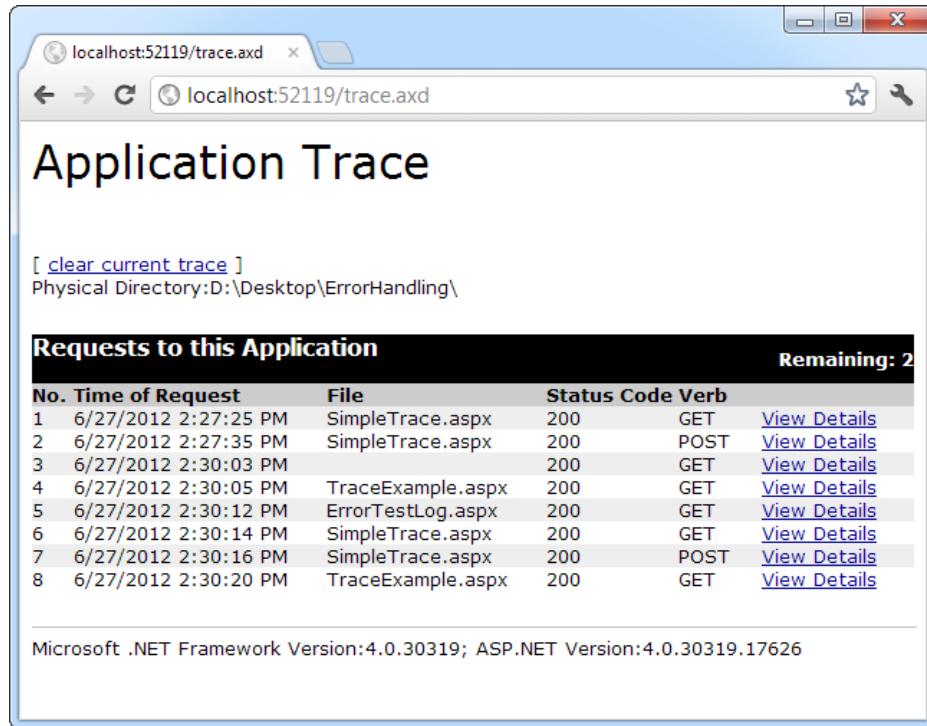


Figure 7-19. Traced application requests

Note Sometimes when requesting trace.axd from Visual Studio's test web server, you may not see the recent traced requests. In this situation, click the browser's Refresh button to get an updated list.

You can see the detailed information for any request by clicking the View Details link. This provides a useful way to store tracing information for a short period of time and allows you to review it without needing to see the actual pages (see Figure 7-20).

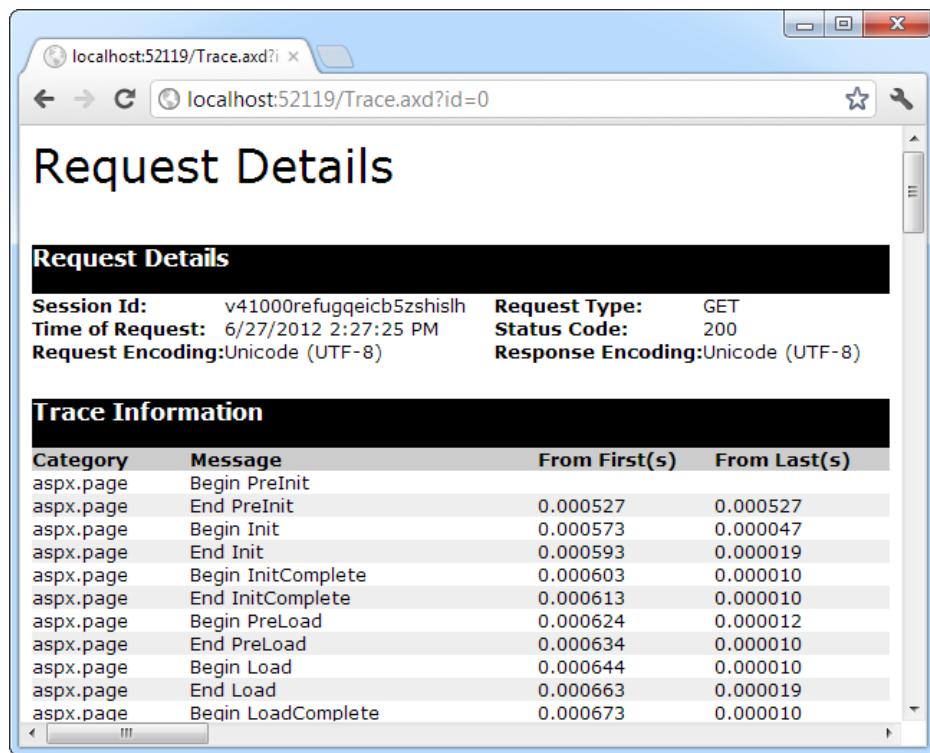


Figure 7-20. One of the traced application requests

The Last Word

One of the most significant differences between an ordinary website and a professional web application is often in how it deals with errors. In this chapter, you learned how to intercept problems with structured error handling, and how to diagnose other, more subtle issues with tracing.



State Management

The most significant difference between programming for the Web and programming for the desktop is *state management*—how you store information over the lifetime of your application. This information can be as simple as a user's name or as complex as a stuffed-full shopping cart for an e-commerce store.

In a traditional desktop application, there's little need to think about state management. Memory is plentiful and always available, and you need to worry about only a single user. In a web application, it's a different story. Thousands of users can simultaneously run the same application on the same computer (the web server), each one communicating over a stateless HTTP connection. These conditions make it impossible to design a web application in the same way as a desktop application.

Understanding these state limitations is the key to creating efficient web applications. In this chapter, you'll see how you can use ASP.NET's state management features to store information carefully and consistently. You'll explore different storage options, including view state, session state, and custom cookies. You'll also consider how to transfer information from page to page by using cross-page posting and the query string.

Understanding the Problem of State

In a traditional desktop application, users interact with a continuously running application. A portion of memory on the desktop computer is allocated to store the current set of working information.

In a web application, the story is quite a bit different. A professional ASP.NET site might look like a continuously running application, but that's really just a clever illusion. In a typical web request, the client connects to the web server and requests a page. When the page is delivered, the connection is severed, and the web server discards all the page objects from memory. By the time the user receives a page, the web page code has already stopped running, and there's no information left in the web server's memory.

This stateless design has one significant advantage. Because clients need to be connected for only a few seconds at most, a web server can handle a huge number of nearly simultaneous requests without a performance hit. However, if you want to retain information for a longer period of time so it can be used over multiple postbacks or on multiple pages, you need to take additional steps.

Using View State

One of the most common ways to store information is in *view state*. View state uses a hidden field that ASP.NET automatically inserts in the final, rendered HTML of a web page. It's a perfect place to store information that's used for multiple postbacks in a single web page.

In Chapter 5, you learned how web controls use view state to keep track of certain details. For example, if you change the text of a label, the Label control automatically stores its new text in view state. That way, the text remains in place the next time the page is posted back. Web controls store most of their property values in view state, provided you haven't switched ViewState off (for example, by setting the EnableViewState property of the control or the page to false).

View state isn't limited to web controls. Your web page code can add bits of information directly to the view state of the containing page and retrieve it later after the page is posted back. The type of information you can store includes simple data types and your own custom objects.

The ViewState Collection

The ViewState property of the page provides the current view-state information. This property provides an instance of the StateBag collection class. The StateBag is a dictionary collection, which means every item is stored in a separate "slot" using a unique string name, which is also called the *key name*.

For example, consider this code:

```
// The this keyword refers to the current Page object. It's optional.  
this.ViewState["Counter"] = 1;
```

This places the value 1 (or rather, an integer that contains the value 1) into the ViewState collection and gives it the descriptive name Counter. If currently no item has the name Counter, a new item will be added automatically. If an item is already stored under the name Counter, it will be replaced.

When retrieving a value, you use the key name. You also need to cast the retrieved value to the appropriate data type by using the casting syntax you saw in Chapter 2 and Chapter 3. This extra step is required because the ViewState collection stores all items as basic objects, which allows it to handle many different data types.

Here's the code that retrieves the counter from view state and converts it to an integer:

```
int counter;  
counter = (int)this.ViewState["Counter"];
```

Note ASP.NET provides many collections that use the same dictionary syntax. These include the collections you'll use for session and application state, as well as those used for caching and cookies. You'll see several collections in this chapter.

A View-State Example

The following example is a simple counter program that records the number of times a button is clicked. Without any kind of state management, the counter will be locked perpetually at 1. With careful use of view state, the counter works as expected.

```
public partial class SimpleCounter : System.Web.UI.Page  
{  
    protected void cmdIncrement_Click(object sender, EventArgs e)  
    {  
        int counter;  
        if (ViewState["Counter"] == null)  
        {  
            counter = 1;  
        }  
        else  
        {  
            counter = (int)ViewState["Counter"] + 1;  
        }  
    }  
}
```

```

ViewState["Counter"] = counter;
lblCount.Text = "Counter: " + counter.ToString();
}
}

```

The code checks to make sure the item exists in view state before it attempts to retrieve it. Otherwise, you could easily run into problems such as the infamous *null reference exception* (which is described in Chapter 7).

Figure 8-1 shows the output for this page.

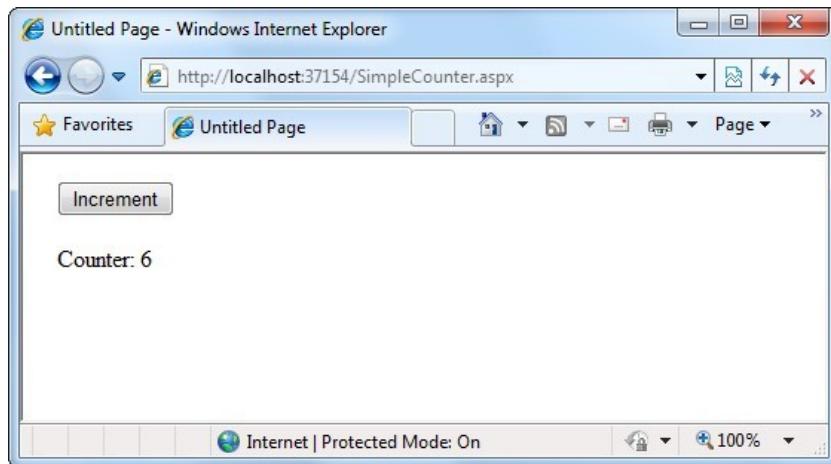


Figure 8-1. A simple view-state counter

Making View State Secure

You probably remember from Chapter 5 that view-state information is stored in a single jumbled string that looks like this:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="dDw3NDg2NTI5MDg7Oz4=" />
```

As you add more information to view state, this value can become much longer. Because this value isn't formatted as clear text, many ASP.NET programmers assume that their view-state data is encrypted. It isn't. Instead, the view-state information is simply patched together in memory and converted to a *Base64 string* (which is a special type of string that's always acceptable in an HTML document because it doesn't include any extended characters). A clever hacker could reverse-engineer this string and examine your view-state data in a matter of seconds.

Fortunately, ASP.NET has the tools to make view state more secure. By default, it makes view state tamper-proof (see the following section, "Tamper-Proof View State") and gives you the additional option of making it secret (see the section after that, "Private View State").

Tamper-Proof View State

ASP.NET uses a hash code to make sure your view-state information can't be altered without your knowledge. Technically, a *hash code* is a cryptographically strong checksum. The idea is that ASP.NET examines all the data

in view state, just before it renders the final page. It runs this data through a hashing algorithm (with the help of a secret key value). The hashing algorithm creates a short segment of data, which is the hash code. This code is then added at the end of the view-state data, in the final HTML that's sent to the browser.

The hash code becomes very useful when the page is posted back to the server. At this point, ASP.NET examines the view-state data and recalculates the hash code by using the same process it used to create the hash code in the first place. ASP.NET then checks whether the newly calculated hash code matches the original hash code, which is stored in the view state for the page. If a malicious user changes part of the view-state data, the new hash code won't match the original value. At this point, ASP.NET will reject the postback completely and show an error page. (You might think a really clever user could get around this by generating fake view-state information *and* a matching hash code. However, malicious users can't generate the right hash code, because they don't have the same cryptographic key as ASP.NET.)

Hash codes are enabled by default, so you don't need to take any extra steps to get this functionality. (In theory, you can disable it, but no one ever does—and you shouldn't.)

Private View State

Even when you use hash codes, the view-state data will still be readable by the user. In many cases, this is completely acceptable—after all, the view state tracks information that's often provided directly through other controls. However, if your view state contains some information you want to keep secret, you can enable view-state *encryption*.

You can turn on encryption for an individual page by using the `ViewStateEncryptionMode` property of the `Page` directive:

```
<%@Page ViewStateEncryptionMode="Always" %>
```

Or you can set the same attribute in a configuration file to configure view-state encryption for all the pages in your website:

```
<configuration>
  ...
  <system.web>
    <pages ViewStateEncryptionMode="Always" />
  ...
</system.web>
</configuration>
```

Either way, this enforces encryption. You have three choices for your view-state encryption setting—always encrypt (Always), never encrypt (Never), or encrypt only if a control specifically requests it (Auto). The default is Auto, which means that the page won't encrypt its view state unless a control on that page specifically requests it. (Technically, a control makes this request by calling the `Page.RegisterRequiresViewStateEncryption()` method.) If no control calls this method to indicate that it has sensitive information, the view state is not encrypted, thereby saving the encryption overhead. On the other hand, a control doesn't have absolute power—if it calls `Page.RegisterRequiresViewStateEncryption()` and the encryption mode is Never, the view state won't be encrypted.

Tip Don't encrypt view-state data if you don't need to do so. The encryption will impose a performance hit because the web server needs to perform the encryption and decryption with each postback.

Retaining Member Variables

You have probably noticed that any information you set in a member variable for an ASP.NET page is automatically abandoned when the page processing is finished and the page is sent to the client. Interestingly, you can work around this limitation by using view state.

The basic principle is to save all member variables to view state when the `Page.PreRender` event occurs, and retrieve them when the `Page.Load` event occurs. Remember, the `Load` event happens every time the page is created. In the case of a postback, the `Load` event occurs first, followed by any other control events.

The following example uses this technique with a single member variable (named `Contents`). The page provides a text box and two buttons. The user can choose to save a string of text and then restore it at a later time (see Figure 8-2). The `Button.Click` event handlers store and retrieve this text by using the `Contents` member variable. These event handlers don't need to save or restore this information by using view state, because the `PreRender` and `Load` event handlers perform these tasks when page processing starts and finishes.

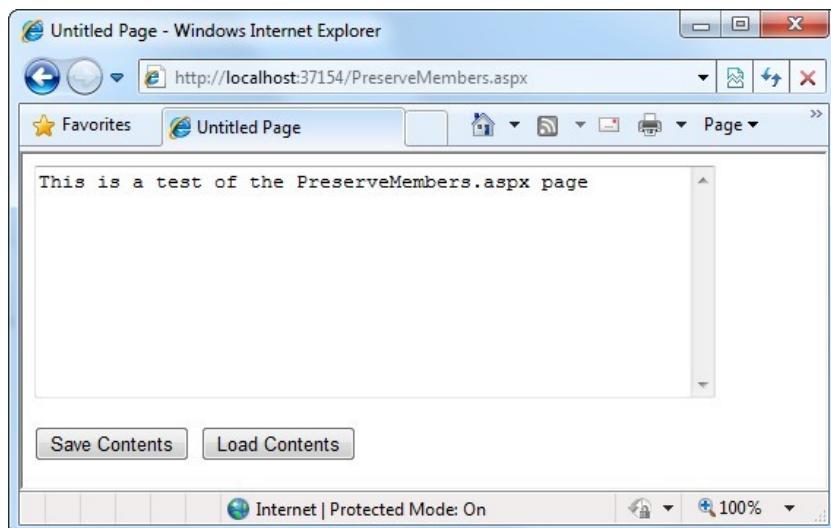


Figure 8-2. A page with state

```
public partial class PreserveMembers : Page
{
    // A member variable that will be cleared with every postback.
    private string contents;

    protected void Page_Load(Object sender, EventArgs e)
    {
        if (this.IsPostBack)
        {
            // Restore variables.
            contents = (string)ViewState["contents"];
        }
    }
}
```

```

protected void Page_PreRender(Object sender, EventArgs e)
{
    // Persist variables.
    ViewState["contents"] = contents;
}

protected void cmdSave_Click(Object sender, EventArgs e)
{
    // Transfer contents of text box to member variable.
    contents = txtValue.Text;
    txtValue.Text = "";
}

protected void cmdLoad_Click(Object sender, EventArgs e)
{
    // Restore contents of member variable to text box.
    txtValue.Text = contents;
}

```

The logic in the Load and PreRender event handlers allows the rest of your code to work more or less as it would in a desktop application. However, you must be careful not to store needless amounts of information when using this technique. If you store unnecessary information in view state, it will enlarge the size of the final page output and can thus slow down page transmission times. Another disadvantage with this approach is that it hides the low-level reality that every piece of data must be explicitly saved and restored. When you hide this reality, it's more likely that you'll forget to respect it and design for it.

If you decide to use this approach to save member variables in view state, use it *exclusively*. In other words, refrain from saving some view-state variables at the PreRender stage and others in control event handlers, because this is sure to confuse you and any other programmer who looks at your code.

Tip The previous code example reacts to the Page.PreRender event, which occurs just after page processing is complete and just before the page is rendered in HTML. This is an ideal place to store any leftover info that is required. You cannot store view-state information in an event handler for the Page.Unload event. The code will not cause an error, the information will not be stored in view state, because the final HTML page has already rendered.

Storing Custom Objects

You can store your own objects in view state just as easily as you store numeric and string types. However, to store an item in view state, ASP.NET must be able to convert it into a stream of bytes so that it can be added to the hidden input field in the page. This process is called *serialization*. If your objects aren't serializable (and by default they're not), you'll receive an error message when you attempt to place them in view state.

To make your objects serializable, you need to add a `[Serializable]` attribute before your class declaration. For example, here's an exceedingly simple Customer class:

```

[Serializable]
public class Customer
{
    private string firstName;

```

```

public string FirstName
{
    get { return firstName; }
    set { firstName = value; }
}

private string lastName;
public string LastName
{
    get { return lastName; }
    set { lastName = value; }
}

public Customer(string firstName, string lastName)
{
    FirstName = firstName;
    LastName = lastName;
}
}

```

Because the Customer class is marked as serializable, it can be stored in view state:

```

// Store a customer in view state.
Customer cust = new Customer("Marsala", "Simons");
ViewState["CurrentCustomer"] = cust;

```

Remember, when using custom objects, you'll need to cast your data when you retrieve it from view state.

```

// Retrieve a customer from view state.
Customer cust;
cust = (Customer)ViewState["CurrentCustomer"];

```

Once you understand this principle, you'll also be able to determine which .NET objects can be placed in view state. You simply need to find the class information in the Visual Studio Help. The easiest approach is to look up the class in the index. For example, to find out about the FileInfo class (which you'll learn about in Chapter 17), look for the index entry *FileInfo class*. In the class documentation, you'll see the declaration for that class, which looks something like this:

```

[Serializable]
[ComVisible(true)]
public sealed class FileInfo : FileSystemInfo

```

If the class declaration is preceded with the Serializable attribute (as it is here), instances of this class can be placed in view state. If the Serializable attribute isn't present, the class isn't serializable, and you won't be able to place instances of it in view state.

Transferring Information Between Pages

One of the most significant limitations with view state is that it's tightly bound to a specific page. If the user navigates to another page, this information is lost. This problem has several solutions, and the best approach depends on your requirements.

In this section, you'll learn two basic techniques to transfer information between pages: cross-page posting and the query string.

Cross-Page Posting

A *cross-page postback* is a technique that extends the postback mechanism you've already learned about so that one page can send the user to another page, complete with all the information for that page. This technique sounds conceptually straightforward, but it's a potential minefield. If you're not careful, it can lead you to create pages that are tightly coupled to others and difficult to enhance and debug.

The infrastructure that supports cross-page postbacks is a property named PostBackUrl, which is defined by the IButtonControl interface and turns up in button controls such as ImageButton, LinkButton, and Button. To use cross-posting, you simply set PostBackUrl to the name of another web form. When the user clicks the button, the page will be posted to that new URL with the values from all the input controls on the current page. (This posted-back information includes the hidden view-state field. As you'll see shortly, it allows ASP.NET to create an up-to-date instance of the source page in memory.)

Here's an example—a page named CrossPage1.aspx that defines a form with two text boxes and a button. When the button is clicked, it posts to a page named CrossPage2.aspx.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="CrossPage1.aspx.cs"
  Inherits="CrossPage1" %>
<html = "http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>CrossPage1</title>
</head>
<body>
  <form id="form1" runat="server" >
    <div>
      First Name:
      <asp:TextBox ID="txtFirstName" runat="server"></asp:TextBox>
      <br />
      Last Name:
      <asp:TextBox ID="txtLastName" runat="server"></asp:TextBox>
      <br />
      <br />
      <asp:Button runat="server" ID="cmdPost"
       PostBackUrl="CrossPage2.aspx" Text="Cross-Page Postback" /><br />
    </div>
  </form>
</body>
</html>
```

The CrossPage1 page doesn't include any code. Figure 8-3 shows how it appears in the browser.

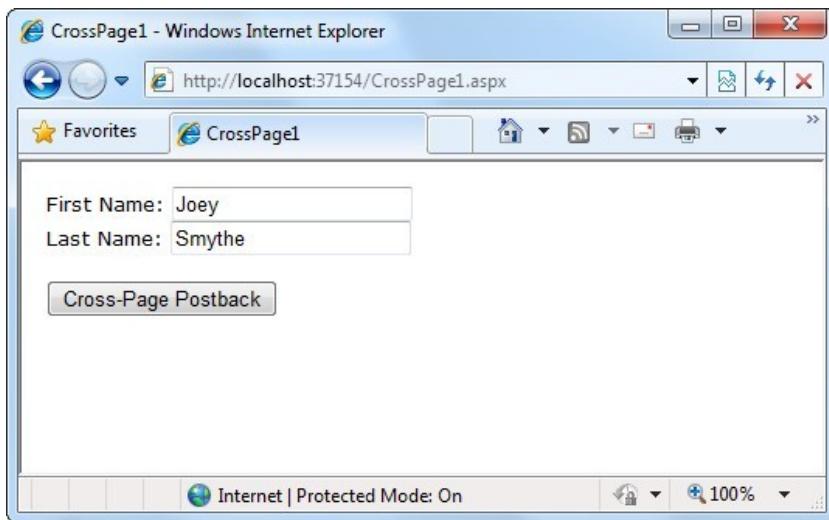


Figure 8-3. The starting point of a cross-page postback

Now if you load this page and click the button, the page will be posted back to CrossPage2.aspx. At this point, the CrossPage2.aspx page can interact with CrossPage1.aspx by using the Page.PreviousPage property. Here's the code for the CrossPage2 page, which includes an event handler that grabs the title from the previous page and displays it:

```
public partial class CrossPage2 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (PreviousPage != null)
        {
            lblInfo.Text = "You came from a page titled " +
                PreviousPage.Title;
        }
    }
}
```

Note that this page checks for a null reference before attempting to access the PreviousPage object. If it's a null reference , no cross-page postback took place. This means CrossPage2.aspx was requested directly or CrossPage2.aspx posted back to itself. Either way, no PreviousPage object is available.

Figure 8-4 shows what you'll see when CrossPage1.aspx posts to CrossPage2.aspx.

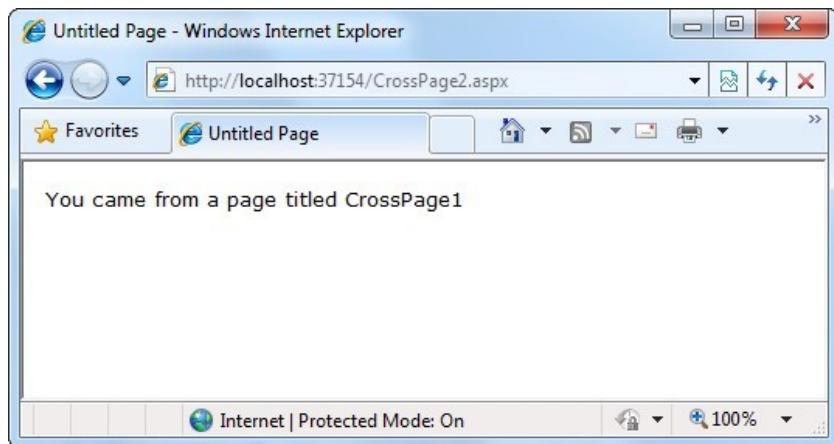


Figure 8-4. The target of a cross-page postback

Getting More Information from the Source Page

The previous example shows an interesting initial test, but it doesn't really allow you to transfer any useful information. After all, you're probably interested in retrieving specific details (such as the text in the text boxes of CrossPage1.aspx) from CrossPage2.aspx. The title alone isn't very interesting.

To get more-specific details, such as control values, you need to cast the PreviousPage reference to the appropriate page class (in this case, it's the CrossPage1 class). Here's an example that handles this situation properly, by checking first whether the PreviousPage object is an instance of the expected class:

```
public partial class CrossPage1 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        CrossPage1 prevPage = PreviousPage as CrossPage1;
        if (prevPage != null)
        {
            // (Read some information from the previous page.)
        }
    }
}
```

Note In a projectless website, Visual Studio may flag this code as an error, indicating that it does not have type information for the source-page class (in this example, that's CrossPage1). However, after you compile your website, the error will disappear.

You can also solve this problem in another way. Rather than casting the reference manually, you can add the PreviousPageType directive to the .aspx page that receives the cross-page postback (in this example, CrossPage2.aspx), right after the Page directive. The PreviousPageType directive indicates the expected type of the page initiating the cross-page postback. Here's an example:

```
<%@ PreviousPageType VirtualPath="~/CrossPage1.aspx" %>
```

Now the PreviousPage property will automatically use the CrossPage1 type. That allows you to skip the casting code and go straight to work using the previous page object, like this:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (PreviousPage != null)
    {
        // (Read some information from the previous page.)
    }
}
```

However, this approach is more fragile because it limits you to a single-page class. You don't have the flexibility to deal more than one page triggering a cross-page postback. For that reason, it's usually more flexible to use the casting approach.

After you've cast the previous page to the appropriate page type, you still won't be able to directly access the control objects it contains. That's because the controls on the web page are not publicly accessible to other classes. You can work around this by using properties.

For example, if you want to expose the values from two text boxes in the source page, you might add a property for each control variable. Here are two properties you could add to the CrossPage1 class to expose its TextBox controls:

```
public TextBox FirstNameTextBox
{
    get { return txtFirstName; }
}
public TextBox LastNameTextBox
{
    get { return txtLastName; }
}
```

However, this usually isn't the best approach. The problem is that it exposes too many details, giving the target page the freedom to read everything from the text in the text box to its fonts and colors. If you need to change the page later to use different input controls, it will be difficult to maintain these properties. Instead, you'll probably be forced to rewrite code in both pages.

A better choice is to define specific, limited methods or properties that extract just the information you need. For example, you might decide to add a FullName property that retrieves just the text from the two text boxes. Here's the full page code for CrossPage1.aspx with this property:

```
public partial class CrossPage1 : System.Web.UI.Page
{
    public string FullName
    {
        get { return txtFirstName.Text + " " + txtLastName.Text; }
    }
}
```

This way, the relationship between the two pages is clear, simple, and easy to maintain. You can probably change the controls in the source page (CrossPage1) without needing to change other parts of your application. For example, if you decide to use different controls for name entry in CrossPage1.aspx, you will be forced to revise the code for the FullName property. However, your changes would be confined to CrossPage1.aspx, and you wouldn't need to modify CrossPage2.aspx at all.

Here's how you can rewrite the code in CrossPage2.aspx to display the information from CrossPage1.aspx:

```

protected void Page_Load(object sender, EventArgs e)
{
    if (PreviousPage != null)
    {
        lblInfo.Text = "You came from a page titled " +
            PreviousPage.Title + "<br />";

        CrossPage1 prevPage = PreviousPage as CrossPage1;
        if (prevPage != null)
        {
            lblInfo.Text += "You typed in this: " + prevPage.FullName;
        }
    }
}

```

Notice that the target page (CrossPage2.aspx) can access the Title property of the previous page (CrossPage1.aspx) without performing any casting. That's because the Title property is defined as part of the base System.Web.UI.Page class, and so every web page includes it. However, to get access to the more specialized FullName property, you need to cast the previous page to the right page class (CrossPage1) or use the PreviousPageType directive that was discussed earlier.

Figure 8-5 shows the new result.

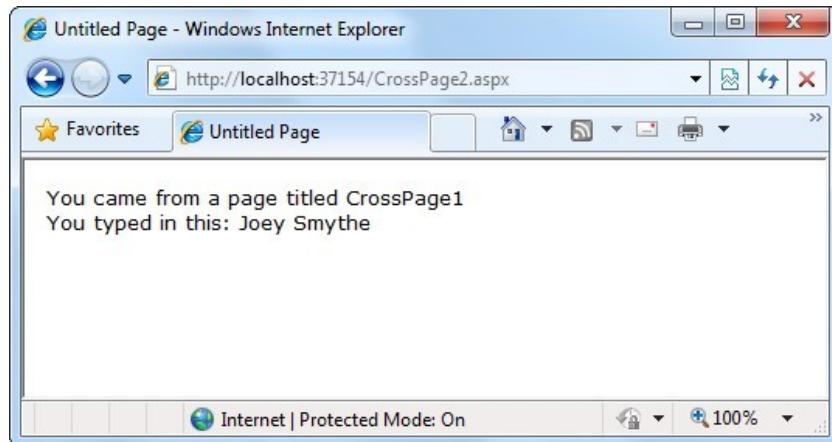


Figure 8-5. Retrieving specific information from the source page

Note Cross-page postbacks are genuinely useful, but they can lead the way to more-complicated pages. If you allow multiple source pages to post to the same destination page, it's up to you to code the logic that figures out which page the user came from and then act accordingly. To avoid these headaches, it's easiest to perform cross-page postbacks between two specific pages only.

ASP.NET uses some interesting sleight of hand to make cross-page postbacks work. The first time the second page accesses Page.PreviousPage, ASP.NET needs to create the previous page object. To do this, it actually starts the page processing but interrupts it just before the PreRender stage, and it doesn't let the page render any HTML output.

However, this still has some interesting side effects. For example, all the page events of the previous page are fired, including Page.Load and Page.Init, and the Button.Click event also fires for the button that triggered the cross-page postback. ASP.NET fires these events because they might be needed to return the source page to the state it was last in, just before it triggered the cross-page postback.

The Query String

Another common approach is to pass information by using a query string in the URL. This approach is commonly found in search engines. For example, if you perform a search on the Google website, you'll be redirected to a new URL that incorporates your search parameters. Here's an example:

```
http://www.google.ca/search?q=organic+gardening
```

The query string is the portion of the URL after the question mark. In this case, it defines a single variable named *q*, which contains the string *organic+gardening*.

The advantage of the query string is that it's lightweight and doesn't exert any kind of burden on the server. However, it also has several limitations:

- Information is limited to simple strings, which must contain URL-legal characters.
- Information is clearly visible to the user and to anyone else who cares to eavesdrop on the Internet.
- The enterprising user might decide to modify the query string and supply new values, which your program won't expect and can't protect against.
- Many browsers impose a limit on the length of a URL (usually from 1 KB to 2 KB). Therefore, you can't place a large amount of information in the query string and still be assured of compatibility with most browsers.

Adding information to the query string is still a useful technique. It's particularly well suited in database applications in which you present the user with a list of items that correspond to records in a database, such as products. The user can then select an item and be forwarded to another page with detailed information about the selected item. One easy way to implement this design is to have the first page send the item ID to the second page. The second page then looks that item up in the database and displays the detailed information. You'll notice this technique in e-commerce sites such as Amazon.com.

To store information in the query string, you first need to place it there. Unfortunately, you have no collection-based way to do this. Instead, you'll need to insert it into the URL yourself. Here's an example that uses this approach with the Response.Redirect() method:

```
// Go to newpage.aspx. Submit a single query string argument
// named recordID, and set to 10.
Response.Redirect("newpage.aspx?recordID=10");
```

You can send multiple parameters as long as they're separated with an ampersand (&):

```
// Go to newpage.aspx. Submit two query string arguments:
// recordID (10) and mode (full).
Response.Redirect("newpage.aspx?recordID=10&mode=full");
```

The receiving page has an easier time working with the query string. It can receive the values from the QueryString dictionary collection exposed by the built-in Request object:

```
string ID = Request.QueryString["recordID"];
```

Note that information is always retrieved as a string, which can then be converted to another simple data type. Values in the QueryString collection are indexed by the variable name. If you attempt to retrieve a value that isn't present in the query string, you'll get a null reference.

Note Unlike view state, information passed through the query string is clearly visible and unencrypted. Don't use the query string for information that needs to be hidden or made tamperproof.

A Query String Example

The next program presents a list of entries. When the user chooses an item by clicking the appropriate item in the list, the user is forwarded to a new page. This page displays the received ID number. This provides a quick and simple query string test with two pages. In a sophisticated application, you would want to combine some of the data control features that are described later in Part 3 of this book.

The first page provides a list of items, a check box, and a submission button (see Figure 8-6).

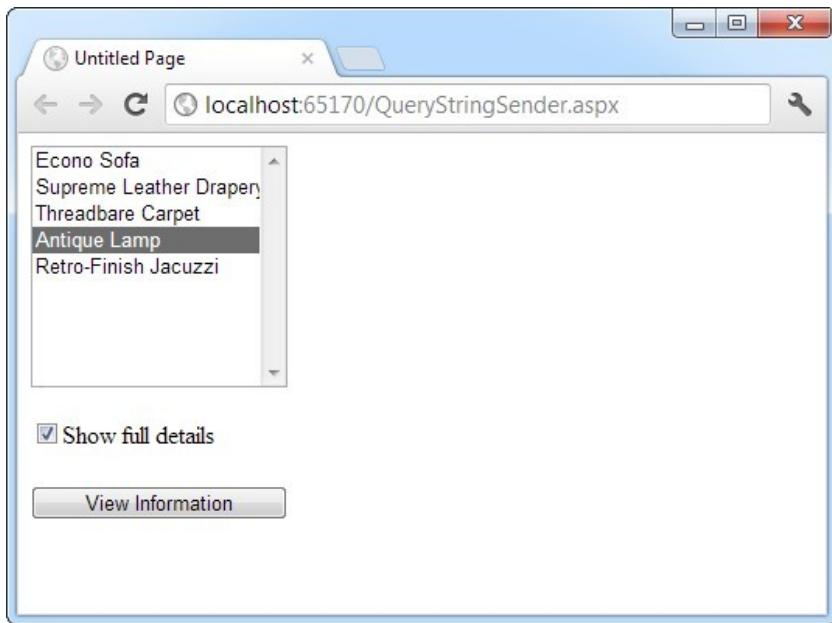


Figure 8-6. A query string sender

Here's the code for the first page:

```
public partial class QueryStringSender : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        if (!this.IsPostBack)
```

```
{  
    // Add sample values.  
    lstItems.Items.Add("Econo Sofa");  
    lstItems.Items.Add("Supreme Leather Drapery");  
    lstItems.Items.Add("Threadbare Carpet");  
    lstItems.Items.Add("Antique Lamp");  
    lstItems.Items.Add("Retro-Finish Jacuzzi");  
}  
}  
  
protected void cmdGo_Click(Object sender, EventArgs e)  
{  
    if (lstItems.SelectedIndex == -1)  
    {  
        lblError.Text = "You must select an item.";  
    }  
    else  
    {  
        // Forward the user to the information page,  
        // with the query string data.  
        string url = "QueryStringRecipient.aspx?";  
        url += "Item=" + lstItems.SelectedItem.Text + "&";  
        url += "Mode=" + chkDetails.Checked.ToString();  
        Response.Redirect(url);  
    }  
}  
}
```

Here's the code for the recipient page (shown in Figure 8-7):



Figure 8-7. A query string recipient

```
public partial class QueryStringRecipient : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        lblInfo.Text = "Item: " + Request.QueryString["Item"];
        lblInfo.Text += "<br />Show Full Record: ";
        lblInfo.Text += Request.QueryString["Mode"];
    }
}
```

One interesting aspect of this example is that it places information in the query string that isn't valid—namely, the space that appears in the item name. When you run the application, you'll notice that ASP.NET encodes the string for you automatically, converting spaces to the valid %20 equivalent escape sequence. The recipient page reads the original values from the QueryString collection without any trouble. This automatic encoding isn't always sufficient. To deal with special characters, you should use the URL encoding technique described in the next section.

URL Encoding

One potential problem with the query string is that some characters aren't allowed in a URL. In fact, the list of characters that are allowed in a URL is much shorter than the list of allowed characters in an HTML document. All characters must be alphanumeric or one of a small set of special characters (including \$-_+!*()'.). Some browsers tolerate certain additional special characters (Internet Explorer is notoriously lax), but many do not. Furthermore, some characters have special meaning. For example, the ampersand (&) is used to separate multiple query string parameters, the plus sign (+) is an alternate way to represent a space, and the number sign (#) is used to point to a specific bookmark in a web page. If you try to send query string values that include any of these characters, you'll lose some of your data. You can test this with the previous example by adding items with special characters in the list box.

To avoid potential problems, it's a good idea to perform *URL encoding* on text values before you place them in the query string. With URL encoding, special characters are replaced by escaped character sequences starting with the percent sign (%), followed by a two-digit hexadecimal representation. For example, the & character becomes %26. The only exception is the space character, which can be represented as the character sequence %20 or the + sign.

To perform URL encoding, you use the `UrlEncode()` and `UrlDecode()` methods of the `HttpServerUtility` class. As you learned in Chapter 5, an `HttpServerUtility` object is made available to your code in every web form through the `Page.Server` property. The following code uses the `UrlEncode()` method to rewrite the previous example, so it works with product names that contain special characters:

```
string url = "QueryStringRecipient.aspx?";
url += "Item=" + Server.UrlEncode(lstItems.SelectedItem.Text) + "&";
url += "Mode=" + chkDetails.Checked.ToString();
Response.Redirect(url);
```

Notice that it's important not to encode everything. In this example, you can't encode the & character that joins the two query string values, because it truly *is* a special character.

You can use the UrlDecode() method to return a URL-encoded string to its initial value. However, you don't need to take this step with the query string. That's because ASP.NET automatically decodes your values when you access them through the Request.QueryString collection. (Many people still make the mistake of decoding the query string values a second time. Usually, decoding already-decoded data won't cause a problem. The only exception occurs when you have a value that includes the + sign. In this case, using UrlDecode() will convert the + sign to a space, which isn't what you want.)

Using Cookies

Cookies provide another way to store information for later use. *Cookies* are small files that are created in the web browser's memory (if they're temporary) or on the client's hard drive (if they're permanent). One advantage of cookies is that they work transparently, without the user being aware that information needs to be stored. They also can be easily used by any page in your application and even be retained between visits, which allows for truly long-term storage. They suffer from some of the same drawbacks that affect query strings—namely, they're limited to simple string information, and they're easily accessible and readable if the user finds and opens the corresponding file. These factors make them a poor choice for complex or private information or large amounts of data.

Some users disable cookies on their browsers, which will cause problems for web applications that require them. Also, users might manually delete the cookie files stored on their hard drives. But for the most part, cookies are widely adopted and used extensively on many websites.

Before you can use cookies, you should import the System.Net namespace so you can easily work with the appropriate types:

using System.Net;

Cookies are fairly easy to use. Both the Request and Response objects (which are provided through Page properties) provide a Cookies collection. The important trick to remember is that you retrieve cookies from the Request object, and you set cookies by using the Response object.

To set a cookie, just create a new HttpCookie object. You can then fill it with string information (using the familiar dictionary pattern) and attach it to the current web response:

```
// Create the cookie object.
HttpCookie cookie = new HttpCookie("Preferences");

// Set a value in it.
cookie["LanguagePref"] = "English";

// Add another value.
cookie["Country"] = "US";

// Add it to the current web response.
Response.Cookies.Add(cookie);
```

A cookie added in this way will persist until the user closes the browser and will be sent with every request. To create a longer-lived cookie, you can set an expiration date:

```
// This cookie lives for one year.
cookie.Expires = DateTime.Now.AddYears(1);
```

You retrieve cookies by cookie name, using the Request.Cookies collection. Here's how you retrieve the preceding cookie, which is named *Preferences*:

```
HttpCookie cookie = Request.Cookies["Preferences"];
```

This code won't cause an exception if the cookie doesn't exist. Instead, you'll simply get a null reference. Before you attempt to retrieve any data from the cookie, you should test the reference to make sure you actually have the cookie:

```
string language;
if (cookie != null)
{
    language = cookie["LanguagePref"];
}
```

The only way to remove a cookie is by replacing it with a cookie that has an expiration date that has already passed. This code demonstrates the technique:

```
HttpCookie cookie = new HttpCookie("Preferences");
cookie.Expires = DateTime.Now.AddDays(-1);
Response.Cookies.Add(cookie);
```

A Cookie Example

The next example shows a typical use of cookies to store a customer name (Figure 8-8). To try this example, begin by running the page, entering a name, and clicking the Create Cookie button. Then close the browser, and request the page again. The second time, the page will find the cookie, read the name, and display a welcome message.

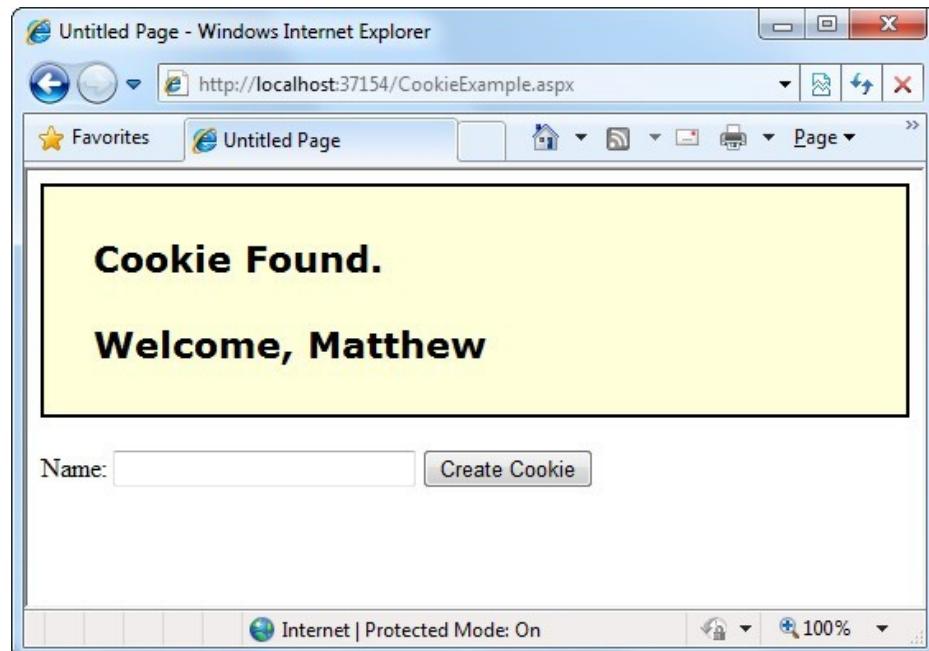


Figure 8-8. Displaying information from a custom cookie

Here's the code for this page:

```
public partial class CookieExample : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        HttpCookie cookie = Request.Cookies["Preferences"];
        if (cookie == null)
        {
            lblWelcome.Text = "<b>Unknown Customer</b>";
        }
        else
        {
            lblWelcome.Text = "<b>Cookie Found.</b><br /><br />";
            lblWelcome.Text += "Welcome, " + cookie["Name"];
        }
    }

    protected void cmdStore_Click(Object sender, EventArgs e)
    {
        // Check for a cookie, and create a new one only if
        // one doesn't already exist.
        HttpCookie cookie = Request.Cookies["Preferences"];
        if (cookie == null)
        {
            cookie = new HttpCookie("Preferences");
        }

        cookie["Name"] = txtName.Text;
        cookie.Expires = DateTime.Now.AddYears(1);
        Response.Cookies.Add(cookie);

        lblWelcome.Text = "<b>Cookie Created.</b><br /><br />";
        lblWelcome.Text += "New Customer: " + cookie["Name"];
    }
}
```

Note You'll find that some other ASP.NET features use cookies. Two examples are session state (which allows you to temporarily store user-specific information in server memory) and forms security (which allows you to protect portions of a website and force users to access it through a login page). Chapter 19 discusses forms security. The next section of this chapter discusses session state.

Managing Session State

There comes a point in the life of most applications when they begin to have more-sophisticated storage requirements. An application might need to store and access complex information such as custom data objects, which can't be easily persisted to a cookie or sent through a query string. Or the application might have stringent security requirements that prevent it from storing information about a client in view state or in a custom cookie. In these situations, you can use ASP.NET's built-in session-state facility.

Session-state management is one of ASP.NET's premiere features. It allows you to store any type of data in memory on the server. The information is protected, because it is never transmitted to the client, and it's uniquely bound to a specific session. Every client that accesses the application has a different session and a distinct collection of information. Session state is ideal for storing information such as the items in the current user's shopping basket when the user browses from one page to another.

Session Tracking

ASP.NET tracks each session by using a unique 120-bit identifier. ASP.NET uses a proprietary algorithm to generate this value, thereby guaranteeing (statistically speaking) that the number is unique and it's random enough that a malicious user can't reverse-engineer or "guess" what session ID a given client will be using. This ID is the only piece of session-related information that is transmitted between the web server and the client.

When the client presents the session ID, ASP.NET looks up the corresponding session, retrieves the objects you stored previously, and places them into a special collection so they can be accessed in your code. This process takes place automatically.

For this system to work, the client must present the appropriate session ID with each request. You can accomplish this in two ways:

Using cookies: In this case, the session ID is transmitted in a special cookie (named ASP.NET_SessionId), which ASP.NET creates automatically when the session collection is used. This is the default.

Using modified URLs: In this case, the session ID is transmitted in a specially modified (or *munged*) URL. This allows you to create applications that use session state with clients that don't support cookies.

Ordinarily, ASP.NET uses cookies to track session state. You'll learn how to switch to the modified-URL system (called cookieless sessions) later in this chapter, when you tackle session-state configuration. But first, it's time to see for yourself how session state works in a website.

Using Session State

You can interact with session state by using the System.Web.SessionState.HttpSessionState class, which is provided in an ASP.NET web page as the built-in Session object. The syntax for adding items to the collection and retrieving them is basically the same as for adding items to a page's view state.

For example, you might store a DataSet in session memory like this:

```
Session["InfoDataSet"] = dsInfo;
```

You can then retrieve it with an appropriate conversion operation:

```
dsInfo = (DataSet)Session["InfoDataSet"];
```

Of course, before you attempt to use the dsInfo object, you'll need to check that it actually exists—in other words, that it isn't a null reference. If the dsInfo is null, it's up to you to regenerate it. (For example, you might decide to query a database to get the latest data.)

Note Chapter 14 explores the DataSet.

Session state is global to your entire application for the current user. However, session state can be lost in several ways:

- If the user closes and restarts the browser.
- If the user accesses the same page through a different browser window, although the session will still exist if a web page is accessed through the original browser window. Browsers differ on how they handle this situation.
- If the session times out due to inactivity. More information about session timeout can be found in the configuration section.
- If your web page code ends the session by calling the Session.Abandon() method.

In the first two cases, the session actually remains in memory on the web server, because ASP.NET has no idea that the client has closed the browser or changed windows. The session will linger in memory, remaining inaccessible, until it eventually expires. (Ordinarily, that's after 20 minutes, but you'll learn how to configure it later in this chapter.)

Table 8-1 describes the key methods and properties of the HttpSessionState class.

Table 8-1. HttpSessionState Members

Member	Description
Count	Provides the number of items in the current session collection.
IsCookieless	Identifies whether the session is tracked with a cookie or modified URLs.
IsNewSession	Identifies whether the session was created only for the current request. If no information is in session state, ASP.NET won't bother to track the session or create a session cookie. Instead, the session will be re-created with every request.
Keys	Gets a collection of all the session keys that are currently being used to store items in the session-state collection.
Mode	Provides an enumerated value that explains how ASP.NET stores session-state information. This storage mode is determined based on the web.config settings discussed in the "Configuring Session State" section later in this chapter.
SessionID	Provides a string with the unique session identifier for the current client.
Timeout	Determines the number of minutes that will elapse before the current session is abandoned, provided that no more requests are received from the client. This value can be changed programmatically, letting you make the session collection longer when needed.
Abandon()	Cancels the current session immediately and releases all the memory it occupied. This is a useful technique in a logoff page to ensure that server memory is reclaimed as quickly as possible.
Clear()	Removes all the session items but doesn't change the current session identifier.

A Session-State Example

The next example uses session state to store several Furniture data objects. The data object combines a few related variables and uses a special constructor so it can be created and initialized in one easy line. Rather than use full property procedures, the class takes a shortcut and uses public member variables so that the code listing remains short and concise. (If you refer to the full code in the downloadable examples, you'll see that it uses property procedures.)

```
public class Furniture
{
    public string Name;
    public string Description;
    public decimal Cost;

    public Furniture(string name, string description,
                    decimal cost)
    {
        Name = name;
        Description = description;
        Cost = cost;
    }
}
```

Three Furniture objects are created the first time the page is loaded, and they're stored in session state. The user can then choose from a list of furniture-piece names. When a selection is made, the corresponding object will be retrieved, and its information will be displayed, as shown in Figure 8-9.

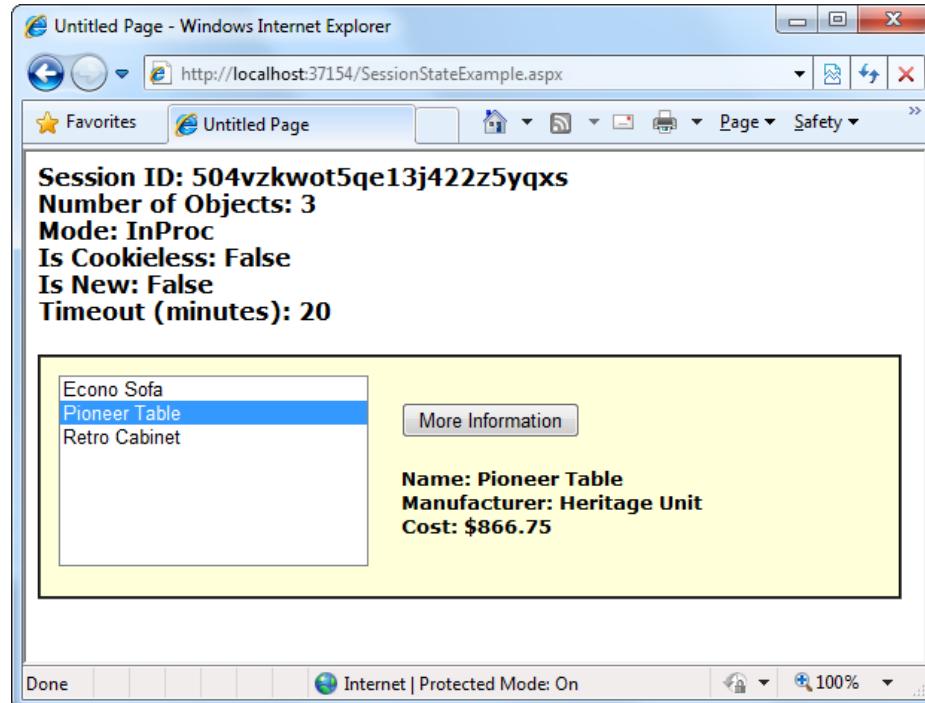


Figure 8-9. A session-state example with data objects

```

public partial class SessionStateExample : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            // Create Furniture objects.
            Furniture piece1 = new Furniture("Econo Sofa",
                "Acme Inc.", 74.99M);
            Furniture piece2 = new Furniture("Pioneer Table",
                "Heritage Unit", 866.75M);
            Furniture piece3 = new Furniture("Retro Cabinet",
                "Sixties Ltd.", 300.11M);

            // Add objects to session state.
            Session["Furniture1"] = piece1;
            Session["Furniture2"] = piece2;
            Session["Furniture3"] = piece3;

            // Add rows to list control.
            lstItems.Items.Add(piece1.Name);
            lstItems.Items.Add(piece2.Name);
            lstItems.Items.Add(piece3.Name);
        }

        // Display some basic information about the session.
        // This is useful for testing configuration settings.
        lblSession.Text = "Session ID: " + Session.SessionID;
        lblSession.Text += "<br />Number of Objects: ";
        lblSession.Text += Session.Count.ToString();
        lblSession.Text += "<br />Mode: " + Session.Mode.ToString();
        lblSession.Text += "<br />Is Cookieless: ";
        lblSession.Text += Session.IsCookieless.ToString();
        lblSession.Text += "<br />Is New: ";
        lblSession.Text += Session.IsNewSession.ToString();
        lblSession.Text += "<br />Timeout (minutes): ";
        lblSession.Text += Session.Timeout.ToString();
    }

    protected void cmdMoreInfo_Click(Object sender, EventArgs e)
    {
        if (lstItems.SelectedIndex == -1)
        {
            lblRecord.Text = "No item selected.";
        }
        else
        {
            // Construct the right key name based on the index.
            string key = "Furniture" +
                (lstItems.SelectedIndex + 1).ToString();
        }
    }
}

```

```

// Retrieve the Furniture object from session state.
Furniture piece = (Furniture)Session[key];

// Display the information for this object.
lblRecord.Text = "Name: " + piece.Name;
lblRecord.Text += "<br />Manufacturer: ";
lblRecord.Text += piece.Description;
lblRecord.Text += "<br />Cost: " + piece.Cost.ToString("c");
}
}
}

```

It's also a good practice to add a few session-friendly features in your application. For example, you could add a logout button to the page that automatically cancels a session by using the Session.Abandon() method. This way, the user will be encouraged to terminate the session rather than just close the browser window, and the server memory will be reclaimed faster. Otherwise, the memory won't be reclaimed until the session times out, which is a potential drag on performance.

Configuring Session State

You configure session state through the web.config file for your current application (which is found in the same virtual directory as the .aspx web page files). The configuration file allows you to set advanced options such as the timeout and the session-state mode.

The following listing shows the most important options that you can set for the <sessionState> element. Keep in mind that you won't use all of these details at the same time. Some settings apply only to certain session-state *modes*, as you'll see shortly.

```

<configuration>
...
<system.web>
...
<sessionState
    cookieless="UseCookies"
    cookieName="ASP.NET_SessionId"
    regenerateExpiredSessionId="false"
    timeout="20"
    mode="InProc"
    stateConnectionString="tcpip=127.0.0.1:42424"
    stateNetworkTimeout="10"
    sqlConnectionString="data source=127.0.0.1;Integrated Security=SSPI"
    sqlCommandTimeout="30"
    allowCustomSqlDatabase="false"
    customProvider=""
    compressionEnabled="false"
/>
</system.web>
</configuration>

```

The following sections describe the most important session-state settings. You'll learn how to change session timeouts, use cookieless sessions, and change the way session information is stored.

Timeout

The timeout setting specifies the number of minutes that ASP.NET will wait, without receiving a request, before it abandons the session.

This setting represents one of the important compromises of session state. If sessions are kept too long, the memory usage of a popular web application will increase, and the performance will decline. Ideally, you will choose a timeframe that is short enough to allow the server to reclaim valuable memory after a client stops using the application but long enough to allow a client to pause and continue a session without losing it.

You can also programmatically change the session timeout in code. For example, if you know a session contains an unusually large amount of information, you may need to limit the amount of time the session can be stored. You would then warn the user and change the Timeout property. Here's a sample line of code that changes the timeout to 10 minutes:

```
Session.Timeout = 10;
```

Cookieless

As you already learned, ASP.NET tracks sessions by using a cookie that it creates and maintains automatically (which shouldn't be confused with the custom cookies your web page may create and manipulate). The ASP.NET session cookie is a bit of behind-the-scenes plumbing, and most of the time you won't think twice about it. But in some environments, cookies aren't the best choice. For example, maybe they're restricted by super-strict security settings. In this case, you may choose to use cookieless sessions.

To allow (or enforce) cookieless sessions, you set the cookieless attribute to one of the values defined by the `HttpCookieMode` enumeration, as listed in Table 8-2.

Table 8-2. *HttpCookieMode Values*

Value	Description
UseCookies	Cookies are always used, even if the browser or device doesn't support cookies or they are disabled. This is the default. If the device does not support cookies, session information will be lost over subsequent requests, because each request will get a new ID.
UseUri	Cookies are never used, regardless of the capabilities of the browser or device. Instead, the session ID is stored in the URL.
UseDeviceProfile	ASP.NET chooses whether to use cookieless sessions by examining the <code>BrowserCapabilities</code> object. The drawback is that this object indicates what the device should support—it doesn't take into account that the user may have disabled cookies in a browser that supports them.
AutoDetect	ASP.NET attempts to determine whether the browser supports cookies by attempting to set and retrieve a cookie (a technique commonly used on the Web). This technique can correctly determine whether a browser supports cookies but has them disabled, in which case cookieless mode is used instead.

Here's an example that forces cookieless mode:

```
<sessionState cookieless="UseUri" ... />
```

In cookieless mode, the session ID will automatically be inserted into the URL. When ASP.NET receives a request, it will remove the ID, retrieve the session collection, and forward the request to the appropriate directory. Figure 8-10 shows a munged URL.



Figure 8-10. A munged URL with the session ID

Because the session ID is inserted in the current URL, relative links also automatically gain the session ID. In other words, if the user is currently stationed on Page1.aspx and clicks a relative link to Page2.aspx, the relative link includes the current session ID as part of the URL. The same is true if you call Response.Redirect() with a relative URL, as shown here:

```
Response.Redirect("Page2.aspx");
```

Figure 8-11 shows a sample website (included with the online samples in the CookielessSessions directory) that tests cookieless sessions. It contains two pages and uses cookieless mode. The first page (Cookieless1.aspx) contains a HyperLink control and two buttons, all of which take you to a second page (Cookieless2.aspx). The trick is that these controls have different ways of performing their navigation. Only two of them work with cookieless sessions—the third loses the current session.

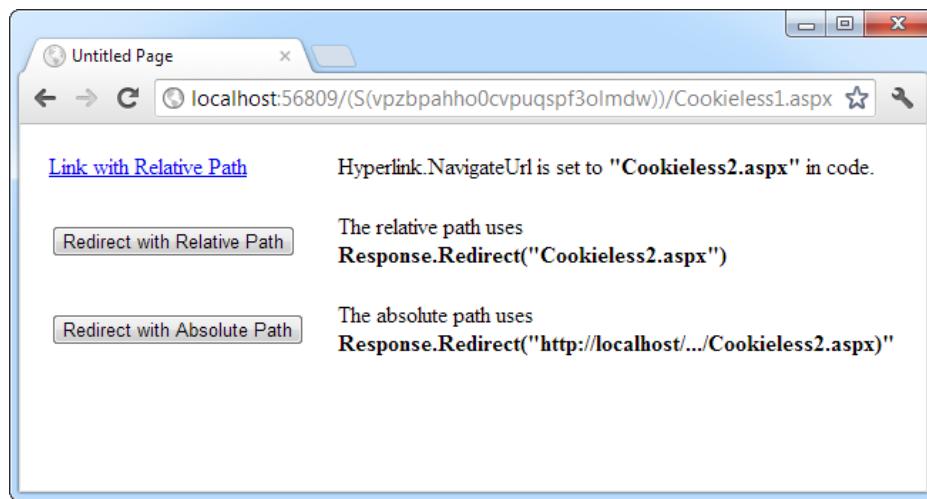


Figure 8-11. Three tests of cookieless sessions

The HyperLink control navigates to the page specified in its NavigateUrl property, which is set to the relative path Cookieless2.aspx. If you click this link, the session ID is retained in the URL, and the new page can retrieve the session information. This demonstrates that cookieless sessions work with relative links.

The two buttons on this page use programmatic redirection by calling the Response.Redirect() method. The first button uses the relative path Cookieless2.aspx, much like the HyperLink control. This approach works with cookieless session state and preserves the munged URL with no extra steps required.

```
protected void cmdLink_Click(Object sender, EventArgs e)
{
    Response.Redirect("Cookieless2.aspx");
}
```

The only real limitation of cookieless state is that you cannot use absolute links (links that include the full URL, starting with http://). The second button uses an absolute link to demonstrate this problem. Because ASP.NET cannot insert the session ID into the URL, the session is lost.

```
protected void cmdLinkAbsolute_Click(Object sender, EventArgs e)
{
    Response.Redirect("http://localhost:56371/CookielessSessions/Cookieless2.aspx");
}
```

Now the target page checks for the session but can't find it. ASP.NET generates a new session ID (which it inserts in the URL), and the original information is lost. Figure 8-12 shows the result.



Figure 8-12. A lost session

Writing the code to demonstrate this problem in a test environment is a bit tricky. The problem is that Visual Studio's integrated web server chooses a different port for your website every time you start it. As a result, you'll need to edit the code every time you open Visual Studio so that your URL uses the right port number (such as 56371 in the previous example).

There's another workaround. You can use some crafty code that gets the current URL from the page and just modifies the last part of it (changing the page name from Cookieless1.aspx to Cookieless2.aspx). Here's how:

```
// Create a new URL based on the current URL (but ending with
// the page Cookieless2.aspx instead of Cookieless1.aspx.
string url = "http://" + Request.Url.Authority +
    Request.Url.Segments[0] + Request.Url.Segments[1] +
    "Cookieless2.aspx";
Response.Redirect(url);
```

Of course, if you deploy your website to a real virtual directory that's hosted by IIS, you won't use a randomly chosen port number anymore, and you won't experience this quirk. Chapter 26 has more about virtual directories and website deployment.

Dealing with expired Session IDs

By default, ASP.NET allows you to reuse a session identifier. For example, if you make a request and your query string contains an expired session, ASP.NET creates a new session and uses that session ID. The problem is that a session ID might inadvertently appear in a public place—such as in a results page in a search engine. This could lead to multiple users accessing the server with the same session identifier, then all joining the same session with the same shared data.

To avoid this potential security risk, you should include the optional `regenerateExpiredSessionId` attribute and set it to true whenever you use cookieless sessions. This way, a new session ID will be issued if a user connects with an expired session ID. The only drawback is that this process also forces the current page to lose all view state and form data, because ASP.NET performs a redirect to make sure the browser has a valid session identifier.

Mode

The remaining session-state settings allow you to configure ASP.NET to use different session-state services, depending on the mode that you choose. The next few sections describe the modes you can choose from.

Note Changing the mode is an advanced configuration task. To do it successfully, you need to understand the environment in which your web application will be deployed. For example, if you're deploying your application to a third-party web host, you need to know whether the host supports other modes before you try to use them. If you're deploying your application to a network server in your own organization, you need to team up with your neighborhood network administrator.

InProc

InProc is the default mode, and it makes the most sense for small websites. It instructs information to be stored in the same process as the ASP.NET worker threads, which provides the best performance but the least durability. If you restart your server, the state information will be lost. (In ASP.NET, application domains can be restarted for a variety of reasons, including configuration changes and updated pages, and when certain thresholds are met. If you find that you're losing sessions *before* the timeout limit, you may want to experiment with a more durable mode.)

InProc mode won't work if you're using a *web farm*, which is a load-balancing arrangement that uses multiple web servers to run your website. In this situation, different web servers might handle consecutive requests from the same user. If the web servers use InProc mode, each one will have its own private collection of session data. The end result is that users will unexpectedly lose their sessions when they travel to a new page or post back the current one.

Note When using the StateServer and SQLServer modes, the objects you store in session state must be serializable. Otherwise, ASP.NET will not be able to transmit the object to the state service or store it in the database. In this chapter, you learned how to create a serializable Customer class for storing in view state.

Off

This setting disables session-state management for every page in the application. This can provide a slight performance improvement for websites that are not using session state.

StateServer

With this setting, ASP.NET will use a separate Windows service for state management. This service runs on the same web server, but it's outside the main ASP.NET process, which gives it a basic level of protection if the ASP.NET process needs to be restarted. The cost is the increased time delay imposed when state information is transferred between two processes. If you frequently access and change state information, this can make for a fairly unwelcome slowdown.

When using the StateServer setting, you need to specify a value for the `stateConnectionString` setting. This string identifies the TCP/IP address of the computer that is running the StateServer service and its port number (which is defined by ASP.NET and doesn't usually need to be changed). This allows you to host the StateServer on another computer. If you don't change this setting, the local server will be used (set as address 127.0.0.1).

Of course, before your application can use the service, you need to start it. The easiest way to do this is to use the Microsoft Management Console (MMC). Here's how:

1. Select Start and type **Computer Management** into the search box.
2. When the Computer Management utility appears, click it.
3. In the Computer Management window, go to the Services and Applications > Services node.
4. Find the service called ASP.NET State Service in the list, as shown in Figure 8-13.

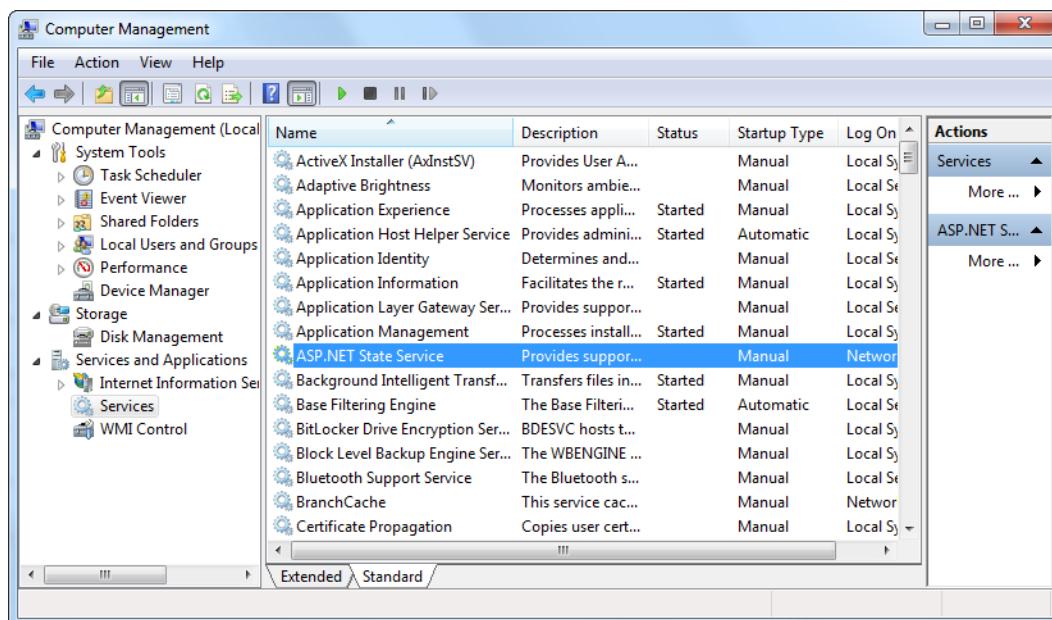


Figure 8-13. The ASP.NET state service

- When you find the service in the list, you can manually start and stop it by right-clicking it. Generally, you'll want to configure Windows to automatically start the service. Right-click it, select Properties, and modify the Startup Type, setting it to Automatic, as shown in Figure 8-14.

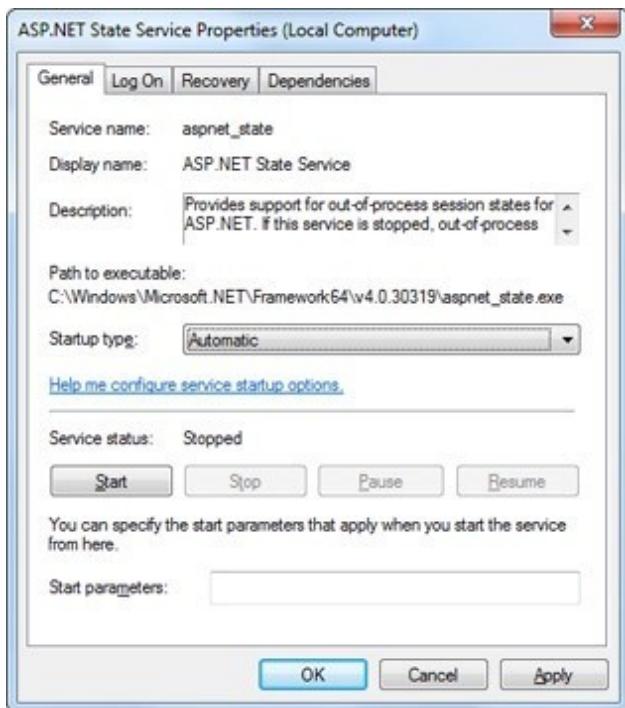


Figure 8-14. Changing the startup type

Note When using StateServer mode, you can also set an optional `stateNetworkTimeout` attribute that specifies the maximum number of seconds to wait for the service to respond before canceling the request. The default value is 10 (seconds).

SQLServer

This setting instructs ASP.NET to use an SQL Server database to store session information, as identified by the `sqlConnectionString` attribute. This is the most resilient state store but also the slowest by far. To use this method of state management, you'll need to have a server with SQL Server installed.

When setting the `sqlConnectionString` attribute, you follow the same sort of pattern you use with ADO.NET data access. Generally, you'll need to specify a data source (the server address) and a user ID and password, unless you're using SQL integrated security.

In addition, you need to install the special stored procedures and temporary session databases. These stored procedures take care of storing and retrieving the session information. ASP.NET includes a command-line tool that does the work for you automatically, called `aspnet_regsql.exe`. It's found in the `c:\Windows\Microsoft.NET\Framework64\v4.0.30319` directory (because 4.0.30319 is the latest version of the CLR, and .NET 4.5 is actually a set of extensions *on top* of that core engine.) To run `aspnet_regsql.exe`, you need to open a Command Prompt window at this location. (One easy way to do that is to browse to the version folder in Windows Explorer, hold down Shift, and right-click the version folder. Then, choose "Open command menu window here" from the menu.) Once you're in the right folder, you can type in an `aspnet_regsql.exe` command.

You can use the `aspnet_regsql.exe` tool to perform several database-related tasks. As you travel through this book, you'll see how to use `aspnet_regsql.exe` with ASP.NET features such as membership (Chapter 20), profiles (Chapter 21), and caching (Chapter 23). To use `aspnet_regsql.exe` to create a session storage database, you supply the `-ssadd` parameter. In addition, you use the `-S` parameter to indicate the database server name, and the `-E` parameter to log in to the database via the currently logged-in Windows user account.

Here's a command that creates the session storage database on the current computer, using the default database name `ASPState`:

```
aspnet_regsql.exe -S localhost -E -ssadd
```

This command uses the alias `localhost`, which tells `aspnet_regsql.exe` to connect to the database server on the current computer.

Note The `aspnet_regsql.exe` command supports additional options that allow you to store session info in a database with a different name. You can find out about these options by referring to the Visual Studio (look up `aspnet_regsql` in the index) or by <http://msdn.microsoft.com/library/ms178586.aspx>. This information also describes the extra steps you need to take to use the database-backed session store with SQL Server Express.

After you've created your session-state database, you need to tell ASP.NET to use it by modifying the `<sessionState>` section of the `web.config` file. If you're using a database named `ASPState` to store your session information (which is the default), you don't need to supply the database name. Instead, you simply have to indicate the location of the server and the type of authentication that ASP.NET should use to connect to it, as shown here:

```
<sessionState mode="SQLServer"
  sqlConnectionString="data source=127.0.0.1;Integrated Security=SSPI"
  ... />
```

When using the `SQLServer` mode, you can also set an optional `sqlCommandTimeout` attribute that specifies the maximum number of seconds to wait for the database to respond before canceling the request. The default is 30 seconds.

Custom

When using custom mode, you need to indicate which session-state store provider to use by supplying the customProvider attribute. The customProvider attribute indicates the name of the class. The class may be part of your web application (in which case the source code is placed in the App_Code subfolder), or it can be in an assembly that your web application is using (in which case the compiled assembly is placed in the Bin subfolder).

Creating a custom state provider is a low-level task that needs to be handled carefully to ensure security, stability, and scalability. Custom state providers are also beyond the scope of this book. However, other vendors may release custom state providers you want to use. For example, Oracle could provide a custom state provider that allows you to store state information in an Oracle database.

Compression

When you set enableCompression to true, session data is compressed before it's passed out of process. The enableCompression setting has an effect only when you're using out-of-process session-state storage, because it's only in this situation that the data is serialized.

To compress and decompress session data, the web server needs to perform additional work. However, this isn't usually a problem, because compression is used in scenarios where web servers have plenty of CPU time to spare but are limited by other factors. Session-state compression makes sense in these two key scenarios:

When storing huge amounts of session-state data in memory: Web server memory is a precious resource. Ideally, session state is used for relatively small chunks of information, while a database deals with the long-term storage of larger amounts of data. But if this isn't the case, and if the out-of-process state server is hogging huge amounts of memory, compression is a potential solution.

When storing session-state data on another computer: In some large-scale web applications, session state is stored out of process (usually in SQL Server) and on a separate computer. As a result, ASP.NET needs to pass the session information back and forth over a network connection. Clearly, this design reduces performance from the speeds you'll see when session state is stored on the web server computer. However, it's still the best compromise for some heavily trafficked web applications with huge session-state storage needs.

The actual amount of compression varies greatly depending on the type of data. However, in testing, Microsoft saw clients achieve 30 percent to 60 percent size reductions, which is enough to improve performance in these specialized scenarios.

Using Application State

Application state allows you to store global objects that can be accessed by any client. Application state is based on the System.Web.HttpApplicationState class, which is provided in all web pages through the built-in Application object.

Application state is similar to session state. It supports the same type of objects, retains information on the server, and uses the same dictionary-based syntax. A common example of using application state is a global counter that tracks the number of times an operation has been performed by all the web application's clients.

For example, you could create a global.asax event handler that tracks the number of sessions that have been created or the number of requests that have been received into the application. Or you can use similar logic in the Page.Load event handler to track the number of times a given page has been requested by various clients. Here's an example of the latter:

```
protected void Page_Load(Object sender, EventArgs e)
{
    // Retrieve the current counter value.
    int count = 0;
    if (Application["HitCounterForOrderPage"] != null)
    {
        count = (int)Application["HitCounterForOrderPage"];
    }

    // Increment the counter.
    count++;

    // Store the current counter value.
    Application["HitCounterForOrderPage"] = count;
    lblCounter.Text = count.ToString();
}
```

Once again, application-state items are stored as objects, so you need to cast them when you retrieve them from the collection. Items in application state never time out. They last until the application or server is restarted or the application domain refreshes itself (because of automatic process recycling settings or an update to one of the pages or components in the application).

Application state isn't often used, because it's generally inefficient. In the previous example, the counter would probably not keep an accurate count, particularly in times of heavy traffic. For example, if two clients requested the page at the same time, you could have a sequence of events like this:

1. User A retrieves the current count (432).
2. User B retrieves the current count (432).
3. User A sets the current count to 433.
4. User B sets the current count to 433.

In other words, one request isn't counted because two clients access the counter at the same time. To prevent this problem, you need to use the Lock() and Unlock() methods, which explicitly allow only one client to access the Application state collection at a time.

```
protected void Page_Load(Object sender, EventArgs e)
{
    // Acquire exclusive access.
    Application.Lock();

    int count = 0;
    if (Application["HitCounterForOrderPage"] != null)
    {
        count = (int)Application["HitCounterForOrderPage"];
    }
    count++;
    Application["HitCounterForOrderPage"] = count;
}
```

```
// Release exclusive access.  
Application.Unlock();  
  
lblCounter.Text = count.ToString();  
}
```

Unfortunately, all other clients requesting the page will be stalled until the Application collection is released. This can drastically reduce performance. Generally, frequently modified values are poor candidates for application state. In fact, application state is rarely used in the .NET world because its two most common uses have been replaced by easier, more efficient methods:

- In the past, application state was used to store application-wide constants, such as a database connection string. As you saw in Chapter 5, this type of constant can be stored in the web.config file, which is generally more flexible because you can change it easily without needing to hunt through web page code or recompile your application.
- Application state can also be used to store frequently used information that is time-consuming to create, such as a full product catalog that requires a database lookup. However, using application state to store this kind of information raises all sorts of problems about how to check whether the data is valid and how to replace it when needed. It can also hamper performance if the product catalog is too large. Chapter 23 introduces a similar but much more sensible approach—storing frequently used information in the ASP.NET cache. Many uses of application state can be replaced more efficiently with caching.

Tip If you decide to use application state, you can initialize its contents when your application first starts by adding the initialization code to the global.asax file in a method named Application_OnStart(), as described in Chapter 23.

Comparing State Management Options

Each state management choice has a different lifetime, scope, performance overhead, and level of support. Table 8-3 and Table 8-4 show an at-a-glance comparison of your state management options.

Table 8-3. State Management Options Compared (Part 1)

	View State	Query String	Custom Cookies
Allowed Data Types	All serializable .NET data types.	A limited amount of string data.	String data.
Storage Location	A hidden field in the current web page.	The browser's URL string.	The client's computer (in memory or a small text file, depending on its lifetime settings).
Lifetime	Retained permanently for postbacks to a single page.	Lost when the user enters a new URL or closes the browser. However, this can be stored in a bookmark.	Set by the programmer. Can be used in multiple pages and can persist between visits.
Scope	Limited to the current page.	Limited to the target page.	The whole ASP.NET application.
Security	Tamperproof by default but easy to read. You can enforce encryption by using the ViewStateEncryptionMode property of the Page directive.	Clearly visible and easy for the user to modify.	Insecure, and can be modified by the user.
Performance Implications	Slow if a large amount of information is stored, but will not affect server performance.	None, because the amount of data is trivial.	None, because the amount of data is trivial.
Typical Use	Page-specific settings.	Sending a product ID from a catalog page to a details page.	Personalization preferences for a website.

Table 8-4. State Management Options Compared (Part 2)

	Session State	Application State
Allowed Data Types	All .NET data types for the default in-process storage mode. All serializable .NET data types if you use an out-of-process storage mode.	All .NET data types.
Storage Location	Server memory, state service, or SQL Server, depending on the mode you choose.	Server memory.
Lifetime	Times out after a predefined period (usually 20 minutes, but can be altered globally or programmatically).	The lifetime of the application (typically, until the server is rebooted).
Scope	The whole ASP.NET application.	The whole ASP.NET application. Unlike other methods, application data is global to all users.
Security	Very secure, because data is never transmitted to the client.	Very secure, because data is never transmitted to the client.
Performance Implications	Slow when storing a large amount of information, especially if there are many users at once, because each user will have their own copy of session data.	Slow when storing a large amount of information, because this data will never time out and be removed.
Typical Use	Storing items in a shopping basket.	Storing any type of global data.

Note ASP.nET has another, more specialized type of state management called *profiles*. Profiles allow you to store and retrieve user-specific information from a database. The only catch is that you need to authenticate in order to get the right information. You'll learn about profiles in Chapter 21.

The Last Word

State management is the art of retaining information between requests. Usually, this information is user-specific (such as a list of items in a shopping cart, a username, or an access level), but sometimes it's global to the whole application (such as usage statistics that track site activity). Because ASP.NET uses a disconnected architecture, you need to explicitly store and retrieve state information with each request. The approach you choose to store this data affects the performance, scalability, and security of your application.

In this chapter, you toured a variety of storage options, including view state, cookies, and session state. You also learned to pass information through cross-page postbacks and the query string. As you develop your own web applications, you can consult Table 8-3 and Table 8-4 to help evaluate different types of state management and determine what's best for your needs.



Styles, Themes, and Master Pages

Using the techniques you've learned so far, you can create polished web pages and let users surf from one page to another. However, to integrate your web pages into a unified, consistent website, you need a few more tools. In this chapter, you'll consider three of the most important tools that you can use: styles, themes, and master pages.

Styles are part of the Cascading Style Sheet (CSS) standard. They aren't directly tied to ASP.NET, but they're still a great help in applying consistent formatting across your entire website. With styles, you can define a set of formatting options once and reuse it to format different elements on multiple pages. You can even create styles that apply their magic automatically—for example, styles that change the font of all the text in your website without requiring you to modify any of the web page code. Best of all, once you've standardized via a specific set of styles and applied them to multiple pages, you can give your entire website a facelift just by editing your style sheet.

Styles are genuinely useful, but there are some things they just can't do. Because styles are based on the HTML standard, they have no understanding of ASP.NET concepts like control properties. To fill the gap, ASP.NET includes a *themes* feature, which plays a similar role as styles but works exclusively with server controls. Much as you use styles to automatically set the formatting characteristics of HTML elements, you use themes to automatically set the properties of ASP.NET controls.

Another feature for standardizing websites is *master pages*. Essentially, a master page is a blueprint for part of your website. Using a master page, you can define web page layout, complete with the usual details such as headers, menu bars, and ad banners. Once you've perfected a master page, you can use it to create *content pages*. Each content page automatically acquires the layout and the content of the linked master page.

By using styles, themes, and master pages, you can ensure that all the pages on your website share a standardized look and layout. In many cases, these details are the difference between an average website and one that looks truly professional.

Styles

In the early days of the Internet, website designers used the formatting features of HTML to decorate these pages. These formatting features were limited, inconsistent, and sometimes poorly supported. Worst of all, HTML formatting led to horribly messy markup, with formatting details littered everywhere.

The solution is the CSS standard, which is supported (with some discrepancies) in all modern browsers. Essentially, CSS gives you a wide range of consistent formatting properties that you can apply to any HTML element. Styles allow you to add borders, set font details, change colors, add margin space and padding, and so on. Many of the examples you've seen so far have in this book have used CSS formatting.

In the following sections, you'll learn the basics of the CSS standard. You'll see how web controls use CSS to apply *their* formatting, and you'll learn how you can explicitly use styles in your ASP.NET web pages.

Note The CSS standard is not a single, universal standard with complete commonality across all browsers. Instead, it exists in several versions (up to CSS3) and includes features that not all browsers support and that browsers implement differently. <http://caniuse.com> is a great resource for figuring out which browser versions support which CSS features, and therefore when it's reasonably "safe" to use a CSS feature on your website. In this chapter, you'll focus on subset of CSS that's well supported, with a high level of consistency across modern browsers.

Style Types

Web pages can use styles in three different ways:

- *Inline style*: An inline style is a style that's placed directly inside an HTML tag. This can get messy, but it's a reasonable approach for one-time formatting. You can remove the style and put it in a style sheet later.
- *Internal style sheet*: An internal style sheet is a collection of styles that are placed in the `<head>` section of your web page markup. You can then use the styles from this style sheet to format the web controls on that page. By using an internal style sheet, you get a clear separation between formatting (your styles) and content (the rest of your HTML markup). You can also reuse the same style for multiple elements.
- *External style sheet*: An external style sheet is similar to an internal style sheet, except it's placed in a completely separate file. This is the most powerful approach because it gives you a way to apply the same style rules to many pages.

You can use all types of styles with ASP.NET web pages. You'll see how in the following sections.

Creating a Basic Inline Style

To apply a style to an ordinary HTML element, you set the `style` attribute. Here's an example that gives a blue background to a paragraph:

```
<p style="background: Blue">This text has a blue background.</p>
```

Every style consists of a list of one or more formatting properties. In the preceding example, the style has a single formatting property, named *background*, which is set to the value Blue. To add multiple style properties, you simply separate them with semicolons, as shown here:

```
<p style = "color:White; background:Blue; font-size:x-large; padding:10px">  
This text has a blue background.</p>
```

This style creates large white text with a blue background and 10 pixels of spacing between the edge of the element (the blue box) and the text content inside.

Note The full list of formatting properties is beyond the scope of this book (although you can get all the details at <http://www.w3schools.com/css>). However, you'll soon see that Visual Studio includes tools that can help you build the styles you want, so you don't need to remember style property names or write styles by hand.

You can use the same approach to apply formatting to a web control using a style; however, you don't need to because web controls provide formatting properties. For example, suppose you create a Label control like this:

```
<asp:Label ID = "MyLabel" runat = "server" ForeColor = "White" BackColor = "Blue"  
Font-Size = "X-Large"> Formatted Text</asp:Label>
```

It's actually rendered into this HTML, which uses an inline style:

```
<span id = "MyLabel" style = "color:White; background-color:Blue; font-size:X-Large">  
Formatted Text</span>
```

Incidentally, if you specify a theme *and* set formatting properties that overlap with your style, the properties have the final say.

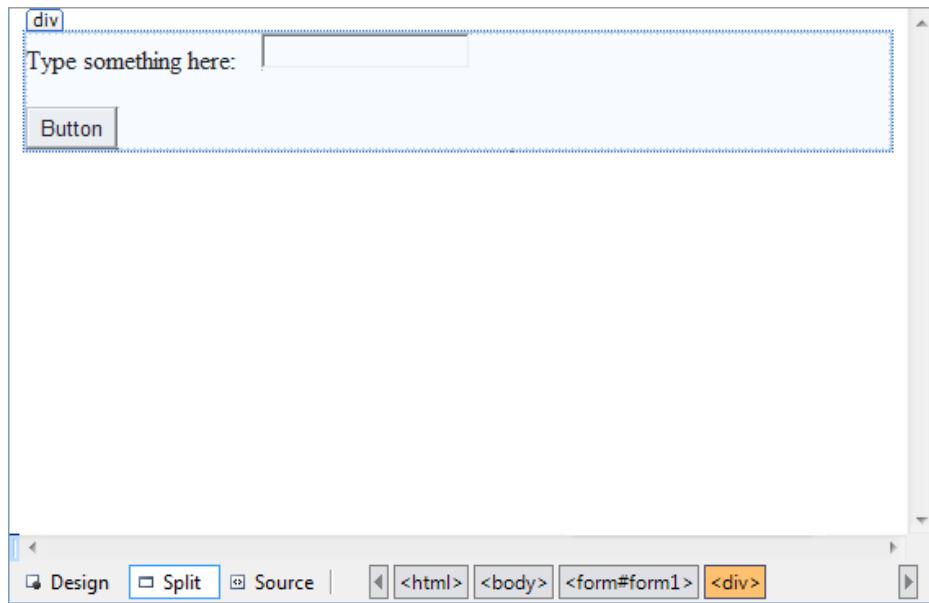
The Style Builder

Visual Studio provides an indispensable style builder that lets you create styles by picking and choosing your style preferences in a dedicated dialog box. To try it out, begin by creating a new page in Visual Studio. Then drop a few controls onto your page (for example, a label, text box, and button).

Every new page starts with an empty `<div>` element. This `<div>` is simply a content container—by default, it doesn't have any appearance. However, by applying style settings to the `<div>`, you can create a bordered content region, and you can change the font and colors of the content inside. In this example, you'll see how to use Visual Studio to build a style for the `<div>` element.

Note CSS supports a feature it calls *inheritance*. With inheritance, some formatting properties (such as family) are passed down from a parent element to other nested elements. In other words, if you set the `font-family` for a `<div>` element, all the elements inside will inherit the same font (unless they explicitly specify otherwise). Other properties, like margin and padding settings, don't use inheritance. To learn more about this behavior and specific properties that use inheritance, you can experiment on your own, consult a dedicated book such as Meyer's *CSS: The Definitive Guide* (O'Reilly), or use the <http://alistapart.com/articles/css>.

For example, imagine you want to create a simple page that includes a few controls, such as the label, text box, and button shown in Figure 12-1. Before formatting the page, make sure all your controls are nested inside the `<div>` element. Your markup should look something like this:



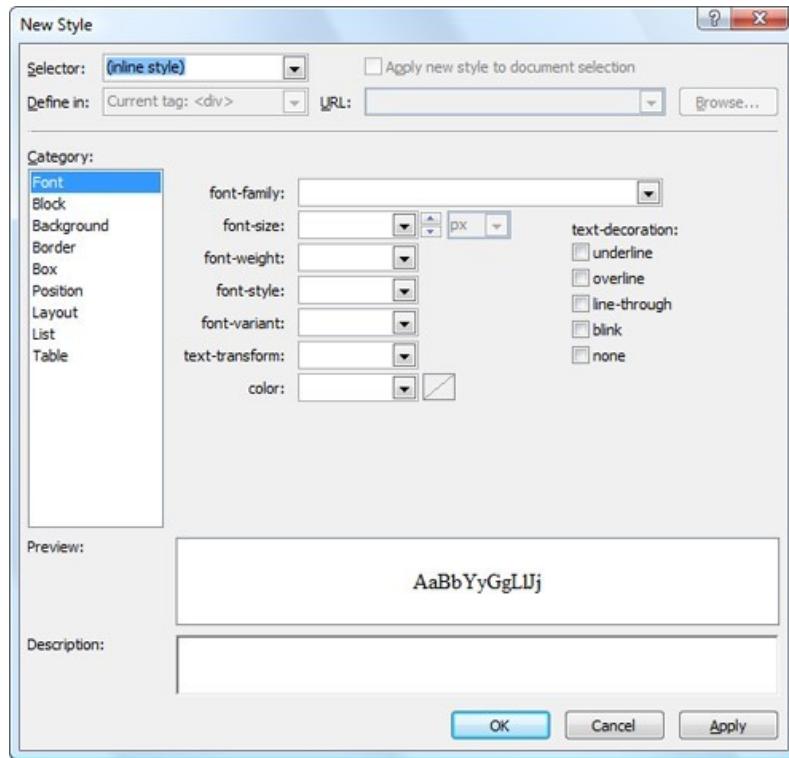


Figure 12-2. Creating an inline style

To specify style settings, you first need to choose one of the categories in the Category list. For example, if you choose Font you'll see a list of font-related formatting settings, such as font family, font size, text color, and so on. You can apply settings from as many different categories as you want. Table 12-1 provides a brief explanation for each category.

Table 12-1. Style Settings in the New Style Dialog Box

Category	Description
Font	Allows you to choose the font family, font size, and text color, and apply other font characteristics (like italics and bold).
Block	Allows you to fine-tune additional text settings, such as the height of lines in a paragraph, the way text is aligned, the amount of indent in the first list, and the amount of spacing between letters and words.
Background	Allows you to set a background color or image.
Border	Allows you to define borders on one or more edges of the element. You can specify the border style, thickness, and color of each edge.
Box	Allows you to define the margin (the space between the edges of the element and its container) and the padding (the space between the edges of the element and its nested content inside).
Position	Allows you to set a fixed width and height for your element, and use absolute positioning to place your element at a specific position on the page. Use these settings with care. When you make your element a fixed size, there's a danger that the content inside can become too big (in which case it leaks out the bottom or the side). When you position your element using absolute coordinates, there's a chance that it can overlap another element.
Layout	Allows you to control miscellaneous layout settings. You can specify whether an element is visible or hidden, whether it floats at the side of the page, and what cursor appears when the user moves the mouse overtop, among other settings.
List	If you're configuring a list (a <code></code> or <code></code> element), you can set the numbering or bullet style. These settings aren't commonly used in ASP.NET web pages because you're more likely to use ASP.NET list controls like the <code>BulletedList</code> .
Table	Allows you to set details that only apply to table elements (such as <code><tr></code> and <code><td></code>). For example, you can control whether borders appear around an empty cell.

Note Remember, you can build a style for any HTML element, not just the `<div>` element. You'll always exactly the same new Style dialog box with the same formatting options.

As you make your selections, Visual Studio shows what your style will look like when applied to some sample text (in the Preview box) and the markup that's needed to define your style (in the Description box). Figure 12-3 shows the New Style dialog box after formatting the `<div>` into a nicely shaded and bordered box. In the Category list, all the categories with formatting settings are highlighted in bold.

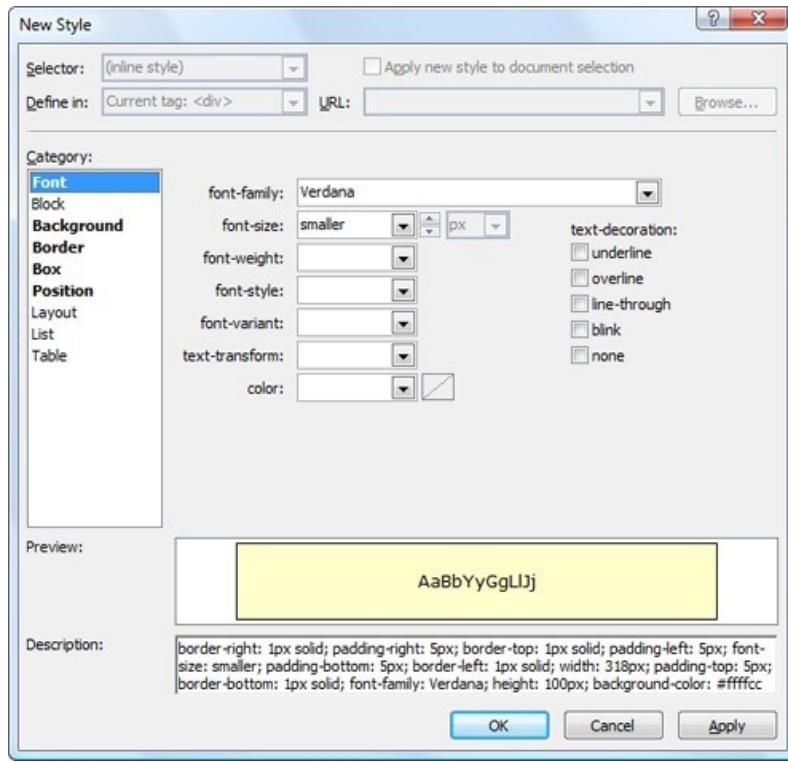
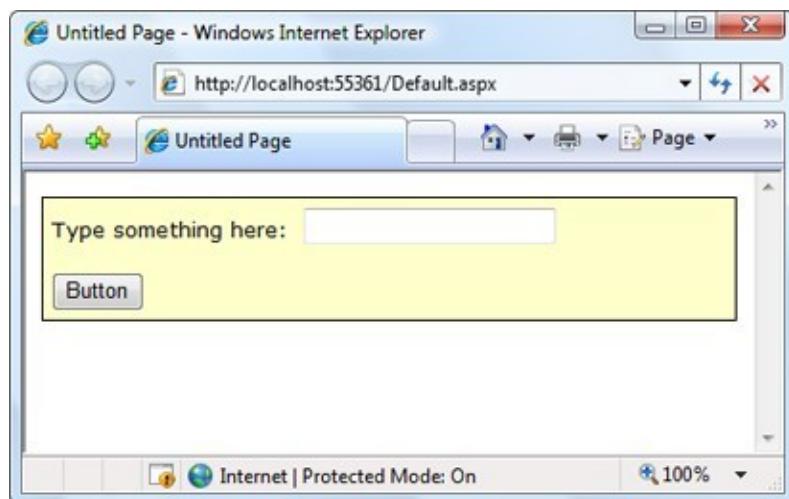


Figure 12-3. Building a styled division

When you click OK, Visual Studio will add the style information to your element. Here's the markup for the formatted `<div>` box:

```
<div style = "border-style: solid; border-color: inherit; border-width: 1px;
padding: 5px; font-size: smaller; font-family: Verdana;
background-color: #ffffcc">
<asp:Label ID = "Label1" runat = "server"> Type something here:</asp:Label>
<asp:TextBox ID = "TextBox1" runat = "server"><br /> <br />
<asp:Button ID = "Button1" runat = "server" Text = "Button">
</asp:Button>
</div>
```

Figure 12-4 shows the final result—a shaded yellow box with a bit of padding and a different font.



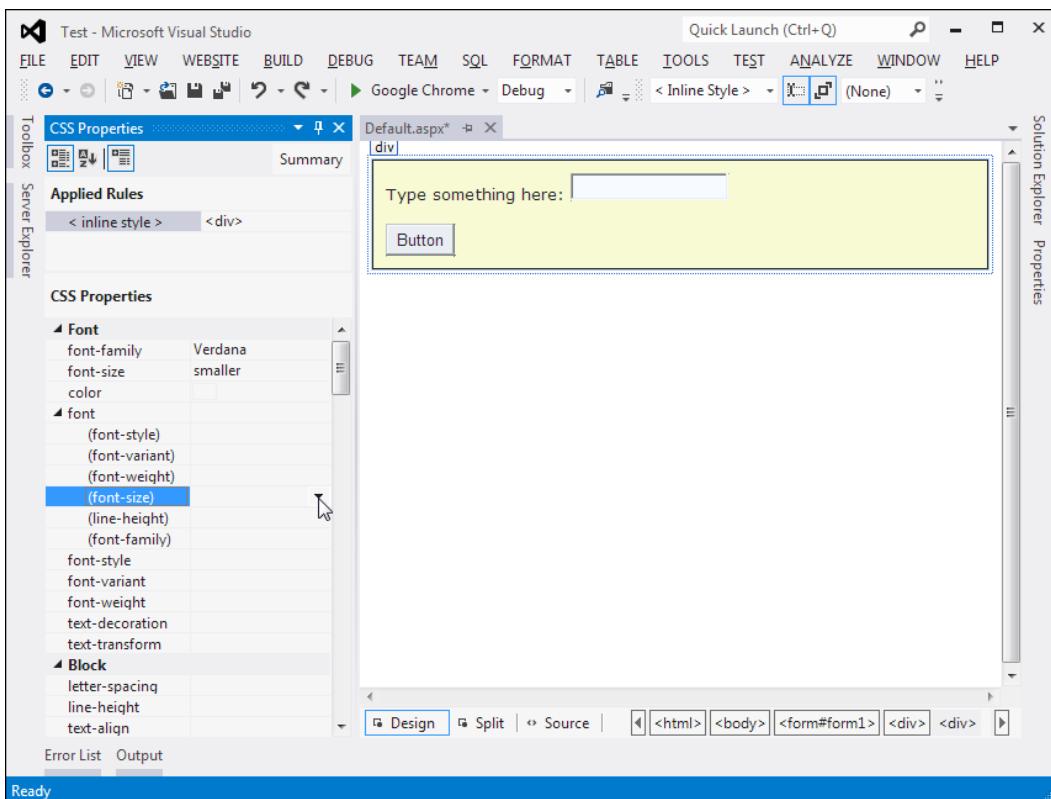


Figure 12-5. The CSS Properties window (on the left)

Once the CSS Properties window is visible, you can use it to view one of your styles. First, find the element or web control that uses the style attribute. Then, click to select it in design view. The style information for that element will appear in the CSS Properties window. For example, Figure 12-5 shows what you'll see for the <div> element that was formatted in the previous section.

The CSS Properties window provides an exhaustive list of all the formatting properties you can use in a style. This list is grouped into categories, and the properties in each category are sorted alphabetically. The ones that are currently set are displayed in bold.

You can use the CSS Properties window to modify existing style properties or set new ones, in the same way that you modify the properties for web controls using the Properties window. For example, in Figure 12-5 the font size is being changed.

Depending on your personal taste, you may find that the CSS Properties window is more convenient than the style builder because it gives you access to every style property at once. Or, you may prefer the more organized views in the style builder. (Your preference might also depend on how much screen real estate you have to devote to the CSS Properties window.) If you decide that you want to return to the style builder to change a style, the process is fairly straightforward. First, select the element that has the inline style. Next, look at the Applied Rules list at the top of the CSS Properties window, which should show the text < inline style >. Right-click that text and choose Modify Style to open the Modify Style dialog box, which looks identical to the New Style dialog box you considered earlier.

Note you can't use the CSS Properties window to create a style. If you select an element that doesn't have a style applied, you won't see anything in the CSS Properties window (unless you select an element that's inside another element and the containing element uses a style).

Style Inheritance

The CSS Properties window is actually a bit more sophisticated than the current discussion has let on. Not only does it show the style for the current element, it also shows any styles that are applied to containing elements. For example, if you look at an element inside the formatted `<div>`, you'll see the style that's applied to the `<div>`. If more than one style is at work (for example, the `<div>` is contained in another formatted `<div>` or you've added a style to the `<body>` element), you'll see all of these styles in the list, with the most general at the top and the most specific at the bottom. You can select the style you want in the list to modify it.

The CSS Properties shows the styles of nested elements because certain style properties are passed down the element tree through inheritance, such as font settings. Other style properties can indirectly affect a nested element—for example, background colors aren't inherited, but because element backgrounds are blank by default, the background of the containing element will show through. It's easy to see this behavior in action—for example, the font and background properties that were applied to the `<div>` element in the previous example affect the formatting of the elements inside.

When displaying inherited styles, the CSS Properties window will sometimes draw a red line through a property name. It does this if the property is set in the parent element but doesn't apply to the nested element. For example, the nested element may override it with its own style, or the property may not be inherited. Figure 12-6 shows an example with a styled `<p>` paragraph inside a styled `<div>`. The style that's inherited from the `<div>` defines the `font-family` property (which is inherited), the `font-size` property (which is inherited but overridden, and is crossed out), and various border properties (which are not inherited, and so are also crossed out).

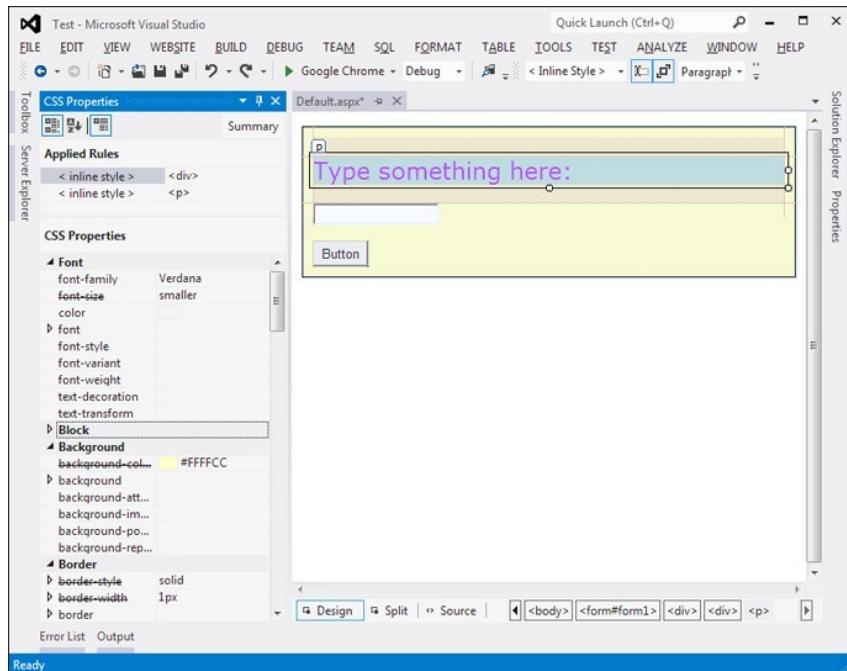


Figure 12-6. Inherited and overridden style properties

Creating a Style Sheet

To really get the value out of CSS, you need to create a style sheet. To create a style sheet in Visual Studio, choose **Website ➤ Add New Item** from the Visual Studio menu. Then, pick the Style Sheet template, specify a file name (like `StyleSheet.css`), and click **Add**.

In a style sheet, you define several styles (also known as *rules*). You can then use these rules to format ordinary HTML and ASP.NET controls. Each rule defines a collection of formatting presets that determines how a single ingredient in your web page should be formatted.

For example, if you want to define a rule for formatting headings, you start by defining a rule with a descriptive name, like this:

```
.heading1
{
}
```

Each rule name has two parts. The portion before the period indicates the HTML element to which the rule applies. In this example, nothing appears before the period, which means the rule can apply to any tag. The portion after the period is a unique name (called the *CSS class name*) that you choose to identify your rule. CSS class names are case sensitive.

Once you've defined a rule, you can add the appropriate formatting information. Here's an example that sets the `heading1` style to a large sans serif font with a green foreground color. The font is set to Verdana (if it's available), or Arial (if it's not), or the browser's default sans serif typeface (if neither Verdana nor Arial is installed).

```
.heading1
{
    font-weight: bold;
    font-size: large;
    color: limegreen;
    font-family: Verdana, Arial, Sans-Serif;
}
```

As you can see, the syntax for defining a style in a style sheet is exactly the same as it is for defining an internal style (not including the rule name and curly braces). By convention, each formatting option in a style is placed on a separate line, but this detail is optional.

You can also create rules that are applied to HTML tags automatically. To do this, specify the tag name for the rule name. Here's a rule that affects all `<h2>` tags on the page that uses the style sheet:

```
h2
{ ... }
```

If you want to employ formatting that applies to the entire web page, you can create a style sheet rule for the `<body>` element:

```
body
{ ... }
```

This gives you a good way to set the default font name and font size.

Fortunately, you don't need to hand-write the rules in your style sheet. Visual Studio allows you to build named styles in a style sheet using the same style builder you used to create inline styles earlier. To use this feature, add a blank style with the right rule name, like this:

```
.myStyle
{
}
```

Then right-click between the two curly braces of an existing style and choose Build Style. You'll see the familiar Modify Style dialog box, where you can point and click your way to custom fonts, borders, backgrounds, and alignment. If you want to create a new style from scratch, simply right-click an empty region of your style sheet and choose Add Style Rule.

A typical style sheet defines a slew of rules. In fact, style sheets are often used to formally define the formatting for every significant piece of a website's user interface. The following style sheet serves this purpose by defining four rules. The first rule sets the font for the `<body>` element, which ensures that the entire page shares a consistent default font. The rest of the rules are class based and need to be applied explicitly to the elements that use them. Two rules define size and color formatting for headings, and the final rule configures the formatting that's needed to create a bordered, shaded box of text.

```
body
{
    font-family: Verdana, Arial, Sans-Serif;
    font-size: small;
}

.heading1
{
    font-weight: bold;
    font-size: large;
    color: lime;
}

.heading2
{
    font-weight: bold;
    font-size: medium;
    font-style: italic;
    color: #C0BA72;
}

.blockText
{
    padding: 10px;
    background-color: #FFFFD9;
    border-style: solid;
    border-width: thin;
}
```

The CSS Outline Window

Visual Studio includes a CSS Outline window that provides an overview of the rules in your style sheet. When you're editing a style sheet, you can show the CSS Outline window by choosing **View** ➤ **Windows** ➤ **Document Outline**. Figure 12-7 shows the outline for the previous style sheet. It clearly indicates that your style sheet includes one element rule (the one that formats the body) and three class rules. You can jump immediately to a specific rule by clicking it in the CSS Outline window.

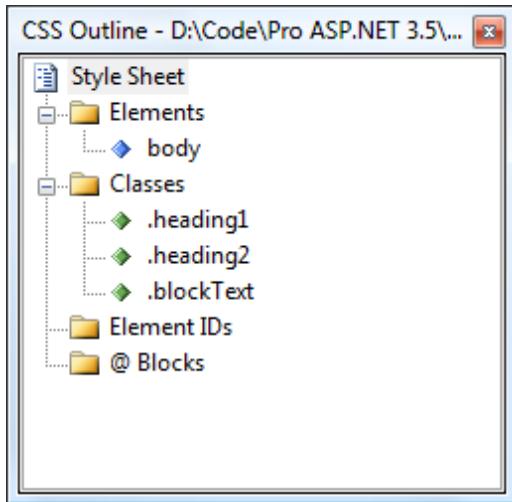


Figure 12-7. Navigating a style sheet with the CSS Outline window

Rule names are technically known as *selectors* because they identify the parts of an HTML document that should be selected for formatting. You've seen how to write selectors that use element types and selectors that use class names. CSS also supports a few more options for building advanced selectors, which aren't described in this chapter. For example, you can create selectors that only apply to a specific element type *inside* a container (for example, headings in a specific `<div>` element). Or, you can create selectors that apply formatting to individual elements that have a specific ID value. (These appear in the CSS Outline window under the Element IDs group.) To learn more about CSS, consult a dedicated book such as *CSS: The Definitive Guide*.

Applying Style Sheet Rules

To use a rule in a web page, you first need to link the page to the appropriate style sheet. You do this by adding a `<link>` element in the `<head>` section of your page. The `<link>` element references the file with the styles you want to use. Here's an example that allows the page to use styles defined in the file `StyleSheet.css`, assuming it's in the same folder as the web page:

```

<html>
  <head runat = "server">
    <title> ...</title>
    <link href = "StyleSheet.css" rel = "stylesheet" />
  </head>

```

```
<body>
...
</body>
</html>
```

There's no reason to attach style sheets by hand. An easier option is to drag your style sheet from the Solution Explorer and drop it onto the design surface of your web page (or the `<head>` section in source view). Visual Studio will insert the `<link>` element you need automatically.

Tip There's one more way to connect a web page to a style sheet in your web application: use the method described in the previous section. When you're editing your web page, switch to design mode and select **Attach Style Sheet**. This shows a dialog box that lists the style sheets in your project. Pick the one you want (often, there's only one), and click OK.

Once you've added the `<link>` element, your style rules are available to use in your web page. You can bind any ordinary HTML element or ASP.NET control to your style rules. For example, if you want an ordinary label to use the heading1 format, set the `Label.CssClass` property to heading1, as shown here:

```
<asp:Label ID = "Label1" runat = "server" Text = "This Label Uses heading1"
CssClass = "heading1"> </asp:Label>
```

You can also set the `CssClass` property from the Properties window, in which case you can choose from the list of styles that are defined in the linked style sheet.

To apply a style to an ordinary piece of HTML, you set the `class` attribute. Here's an example that applies a style to a `<div>` element, which groups together a paragraph of text for easy formatting:

```
<div class = "blockText" id = "paragraph" runat = "server">
<p> This paragraph uses the blockText style.</p>
</div>
```

Incidentally, you can add multiple `<link>` elements to import the styles from several different style sheets. However, you need to be careful that you don't use two style sheets that define styles with the same names.

The Apply Styles Window

Once again, you don't need to edit the markup by hand. You can use Visual Studio's Apply Styles window to attach your style to the elements in a web page. To show the Apply Styles window, open your web page and choose **View** ▶ **Apply Styles**. The Apply Styles window appears on the left with the Toolbox and Server Explorer, as with the other CSS windows you've seen so far.

The Apply Styles window shows a list of all the styles that are available in the attached style sheets, along with a preview of each one (see Figure 12-8). To apply a style, simply select an element on your web page and then click the appropriate style in the Apply Styles window. To remove the current style, choose the first option in the list, which is named Clear Styles.

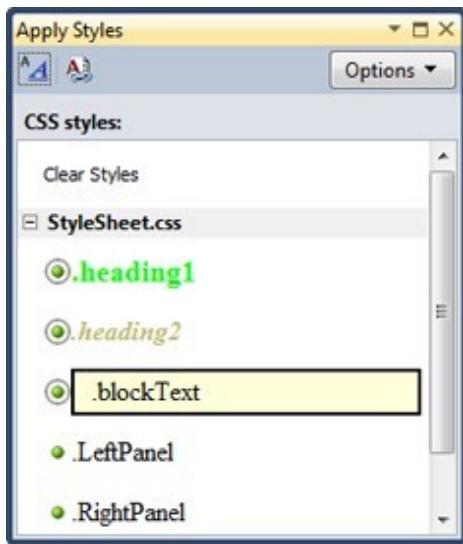


Figure 12-8. Applying a style with the Apply Styles window

Visual Studio is intelligent enough to figure out the appropriate way to apply a style based on what you've selected in your web page:

- If you select a web control, it adds or changes the `CssClass` property.
- If you select an ordinary HTML element, it adds or changes the `class` attribute.
- If you select a section of HTML content, it adds a `` or `<div>` element (depending on the type of content you've selected) and then sets its `class` attribute.

Tip Click the Options button in the Apply Styles window to tweak the way it works. For example, you can preview styles in a different order, or include just those styles that are being used in the current page.

Once you've applied a style, you'll see the result of the new formatting in the design window.

Creating More Styles

In the beginning of this chapter, you saw how you can use the New Style dialog box to create an inline style, which embeds all the formatting in the tag for the affected element (like a `<div>` element). This gives you all the formatting abilities of CSS, but it creates messy markup that you can't reuse. Inline styles are fine for one-off formatting tasks, but they aren't a good choice for formatting "presets" that you want to reuse throughout your website.

Now that you know how to create an external style sheet, it's worth pointing out that the New Style dialog box also works with them. In fact, you can use the New Style dialog box to create a new style in an external style sheet while you're editing a web page, in exactly the same way that you create a new inline style. To do this, follow these steps:

1. Select an element in design view, and choose **Format** **New Style**.
2. In the “Define in” box, choose “Existing style sheet,” and pick the style sheet file from the URL list.
3. In the Selector box, type in a period followed by the name of the style you want to create (like `.heading2`).
4. Finally, configure the formatting, and click OK.

When you follow this process, Visual Studio adds the style information to the linked style sheet. You don’t even need to open the linked CSS file. (Of course, if you still want to create an inline style, no one’s stopping you—just choose “(inline style)” in the Selector box, as usual.)

Using style sheets accomplishes two things. First, it standardizes your layout so that you can quickly format pages without introducing minor mistakes or idiosyncrasies. Second, it separates the formatting information so that it doesn’t appear in your web pages at all, allowing you to modify the format without tracking down each page. And although CSS isn’t a .NET-centric standard, Visual Studio still provides rich support for it.

Themes

With the convenience of CSS styles, you might wonder why developers need anything more. The problem is that CSS rules are limited to a fixed set of style attributes. They allow you to reuse specific formatting details (fonts, borders, foreground and background colors, and so on), but they obviously can’t control other aspects of ASP.NET controls. For example, the CheckBoxList control includes properties that control how it organizes items into rows and columns. Although these properties affect the visual appearance of the control, they’re outside the scope of CSS, so you need to set them by hand. Additionally, you might want to define part of the behavior of the control along with the formatting. For example, you might want to standardize the selection mode of a Calendar control or the wrapping in a TextBox. This obviously isn’t possible through CSS.

The themes feature fills this gap. Like CSS, themes allow you to define a set of style details that you can apply to controls in multiple pages. However, unlike CSS, themes aren’t implemented by the browser. Instead, ASP.NET processes your themes when it creates the page.

Note Themes don’t replace styles. Instead, they complement each other. Styles are particularly useful if you want to apply the same formatting to web controls and ordinary HTML elements. Themes are indispensable if you want to configure control properties that can’t be tailored with CSS.

How Themes Work

All themes are application specific. To use a theme in a web application, you need to create a folder that defines it. This folder needs to be placed in the App_Themes folder, which must be placed inside the top-level directory for your web application. In other words, a web application named SuperCommerce might have a theme named FunkyTheme in the folder SuperCommerce\App_Themes\FunkyTheme. An application can contain definitions for multiple themes as long as each theme is in a separate folder. Only one theme can be active on a given page at a time.

To actually make your theme accomplish anything, you need to create at least one skin file in the theme folder. A *skin file* is a text file with the .skin extension. ASP.NET never serves skin files directly; instead, they’re used behind the scenes to define a theme.

A skin file is essentially a list of control tags—with a twist. The control tags in a skin file don’t need to completely define the control; they need only set the properties that you want to standardize. For example, if you’re trying to apply a consistent color scheme, you might be interested in setting only properties such as ForeColor and BackColor. When you add a control tag for the ListBox in the skin file, it might look like this:

```
<asp:ListBox runat="server" ForeColor="White" BackColor="Orange"/>
```

The `runat="server"` portion is always required. Everything else is optional. You should avoid setting the ID attribute in your skin file because the page that contains the ListBox needs to define a unique name for the control in the actual web page.

It's up to you whether you create multiple skin files or place all your control tags in a single skin file. Both approaches are equivalent because ASP.NET treats all the skin files in a theme directory as part of the same theme definition. Often, it makes sense to put the control tags for complex controls (such as the data controls) in separate skin files. Figure 12-9 shows the relationship between themes and skins in more detail.

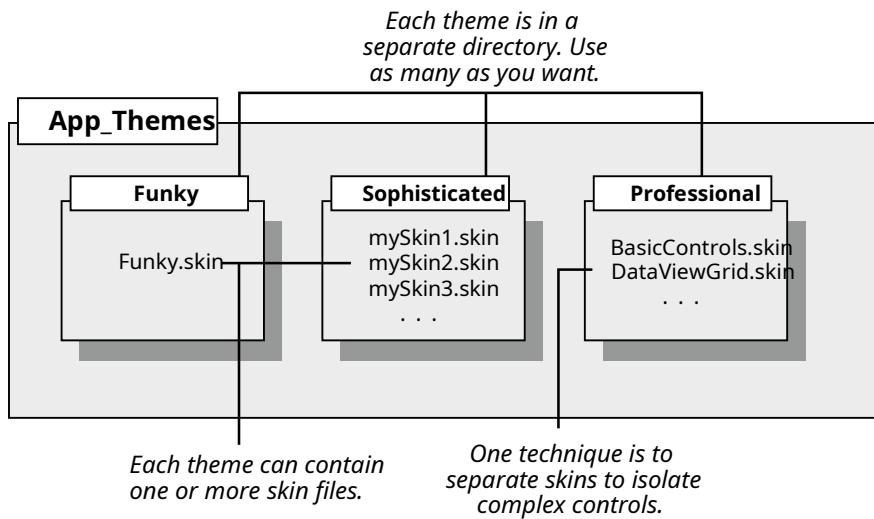


Figure 12-9. Themes and skins

Tip ASP.NET doesn't ship with any predefined themes. This means you'll need to create your own from scratch or download sample themes from website <https://www.asp.net>.

Applying a Simple Theme

To add a theme to your project, select **Website > Add New Item**, and choose Skin File. Visual Studio will warn you that skin files need to be placed in a subfolder of the App_Themes folder and ask you whether that's what you intended. If you choose Yes, Visual Studio will create a folder with the same name as your theme file. You can then rename the folder and the file to whatever you'd like to use. Figure 12-10 shows an example with a theme that contains a single skin file.

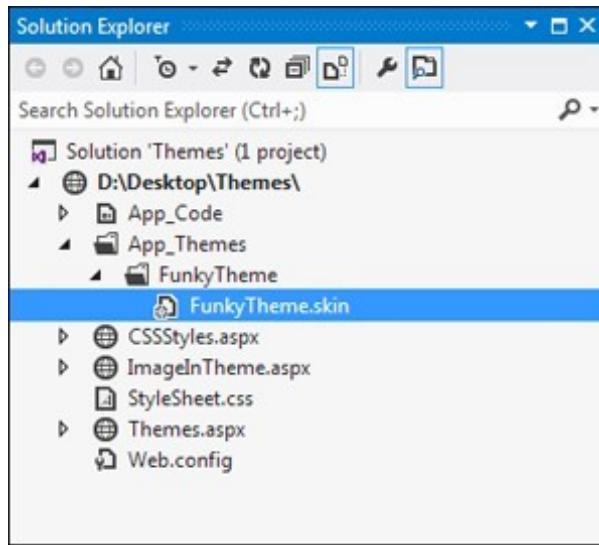


Figure 12-10. A theme in the Solution Explorer

Unfortunately, Visual Studio doesn't include any design-time support for creating themes, so it's up to you to copy and paste control tags from other web pages.

Here's a sample skin file that sets background and foreground colors for several common controls:

```
<asp:ListBox runat = "server" ForeColor = "White" BackColor = "Orange"/>
<asp:TextBox runat = "server" ForeColor = "White" BackColor = "Orange"/>
<asp:Button runat = "server" ForeColor = "White" BackColor = "Orange"/>
```

To apply the theme in a web page, you need to set the Theme attribute of the Page directive to the folder name for your theme.

```
<%@ Page Language = "C#" AutoEventWireup = "true" ... Theme = "FunkyTheme" %>
```

Now ASP.NET will automatically look for a folder named AppThemes\FunkyTheme inside your website, and it will scan all the skin files in that theme.

You can make this change by hand, or you can select the DOCUMENT object in the Properties window at design time and set the Theme property (which provides a handy drop-down list of all your web application's themes). Visual Studio will modify the Page directive accordingly.

When you apply a theme to a page, ASP.NET considers each control on your web page and checks your skin files to see whether they define any properties for that control. If ASP.NET finds a matching tag in the skin file, the information from the skin file overrides the current properties of the control. (And if you have a CSS style that affects the same control, it can override the theme, because the theme is set on the server and the style is applied afterwards, in the browser. However, it's best to avoid this sort of overlapping formatting when possible.)

Figure 12-11 shows the result of applying the FunkyTheme to a simple page. You'll notice that conflicting settings (such as the existing background for the list box) are overwritten. However, changes that don't conflict (such as the custom font for the buttons) are left in place.

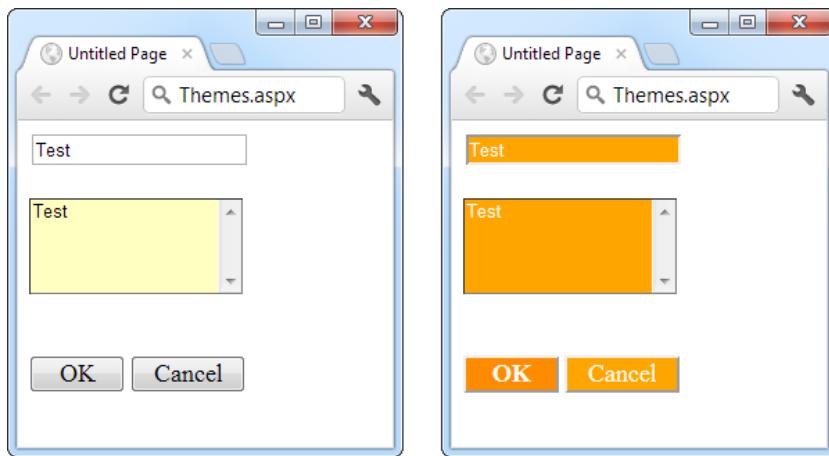


Figure 12-11. A simple page before and after theming

Note This example demonstrates default themes. When you use this approach, your theme settings won't appear in the Visual Studio design environment. That means you won't be able to see the true appearance of the page until you launch it in your browser. If this poses too much of a problem, consider using the SkinId property (described later in the "Creating multiple Skins for the Same Control" section) to explicitly configure each control. When you use this approach, the themed appearance will appear in Visual Studio.

Handling Theme Conflicts

As you've seen, when properties conflict between your controls and your theme, the theme wins. However, in some cases you might want to change this behavior so that your controls can fine-tune a theme by specifically overriding certain details. ASP.NET gives you this option, but it's an all-or-nothing setting that applies to all the controls on the entire page.

To make this change, just use the StyleSheetTheme attribute instead of the Theme attribute in the Page directive. (The StyleSheet designation indicates that this setting works more like CSS.) Here's an example:

```
<%@ Page Language = "C#" AutoEventWireup = "true" ... StyleSheetTheme = "FunkyTheme" %>
```

Now the custom yellow background of the ListBox control takes precedence over the background color specified by the theme. Figure 12-12 shows the result—and a potential problem. Because the foreground color has been changed to white, the lettering in the large text box is now impossible to see. Overlapping formatting specifications can cause glitches like this, which is why it's often better to let your themes take complete control by using the Theme attribute.

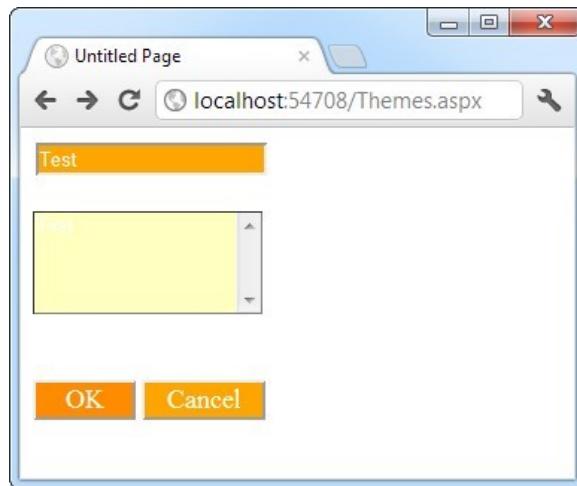


Figure 12-12. Giving the control tag precedence over the theme

Note It's possible to use both the Theme attribute and the StyleSheetTheme attribute at the same time, so that some settings are always applied (those in the Theme attribute) and others are applied only if they are already specified in the control (those in the StyleSheetTheme attribute). However, in practice, this design is confusing and not recommended.

Another option is to configure specific controls so they opt out of the theming process entirely. To do this, simply set the EnableTheming property of the control on the web page to false. ASP.NET will still apply the theme to other controls on the page, but it will skip over the control you've configured.

```
<asp:Button ID = "Button1" runat = "server" ... EnableTheming = "false" />
```

appLYING a theMe to aN eNtire WeBSIte

Using the Page directive, you can bind a theme to a single page. However, you might decide that you want to apply a theme to the entire web application. You can do this by adding the <pages> element in the web.config file for your application, as shown here:

```
<configuration>
<system.web>
  <pages theme = "FunkyTheme">
    ...
  </pages>
</system.web>
</configuration>
```

If you want to use the style sheet behavior so that the theme doesn't overwrite conflicting control properties, use the StyleSheetTheme attribute instead of Theme:

```
<configuration>
<system.web>
  <pages styleSheetTheme = "FunkyTheme">
    ...
  </pages>
</system.web>
</configuration>
```

Either way, when you specify a theme in the web.config file, the theme will be applied throughout the website, provided these pages don't have their own theme settings. If a page specifies the Theme attribute, the page setting will take precedence over the web.config setting. If your page specifies the StyleSheetTheme attribute with a blank string (Theme = ""), no theme will be applied at all. Using this technique, it's just as easy to apply a theme to part of a web application. For example, you can create a separate web.config file for each subfolder and use the <pages> setting to configure different themes.

Creating Multiple Skins for the Same Control

Having each control locked into a single format is great for standardization, but it's probably not flexible enough for a real-world application. For example, you might have several types of text boxes that are distinguished based on where they're used or what type of data they contain. Labels are even more likely to differ, depending on whether they're being used for headings or body text. Fortunately, ASP.NET allows you to create multiple declarations for the same control.

Ordinarily, if you create more than one theme for the same control, ASP.NET will give you a build error stating that you can have only a single default skin for each control. To get around this problem, you need to create a named skin by supplying a SkinID attribute. Here's an example:

```
<asp:ListBox runat = "server" ForeColor = "White" BackColor = "Orange" />
<asp:TextBox runat = "server" ForeColor = "White" BackColor = "Orange" />
<asp:Button runat = "server" ForeColor = "White" BackColor = "Orange" />
<asp:TextBox runat = "server" ForeColor = "White" BackColor = "DarkOrange"
  Font-Bold = "True" SkinID = "Dramatic"/>
<asp:Button runat = "server" ForeColor = "White" BackColor = "DarkOrange"
  Font-Bold = "True" SkinID = "Dramatic"/>
```

The catch is that named skins aren't applied automatically like default skins. To use a named skin, you need to set the SkinID of the control on your web page to match. You can choose this value from a drop-down list that Visual Studio creates based on all your defined skin names, or you can type it in by hand:

```
<asp:Button ID = "Button1" runat = "server" ... SkinID = "Dramatic" />
```

If you don't like the opt-in model for themes, you can make all your skins named. That way, they'll never be applied unless you set the control's SkinID.

ASP.NET is intelligent enough to catch it if you try to use a skin name that doesn't exist, in which case you'll get a build warning. The control will then behave as though you set EnableTheming to false, which means it will ignore the corresponding default skin.

Tip The SkinId doesn't need to be unique. It just has to be unique for each control. For example, imagine you want to create an alternate set of skinned controls that use a slightly smaller font. These controls match the main theme, but they're useful on pages that display a large amount of information. In this case, you can create a separate skin for the ListBox, TextBox, and label controls, and give each one the same skin name (such as Smaller).

More Advanced Skins

So far, the theming examples have applied relatively simple properties. However, you could create much more detailed control tags in your skin file. Most control properties support theming. If a property can't be declared in a theme, you'll receive a build error when you attempt to launch your application.

For example, many controls support styles that specify a range of formatting information. The data controls are one example, and the Calendar control provides another. Here's how you might define Calendar styles in a skin file to match your theme:

```
<asp:Calendar runat = "server" BackColor = "White" ForeColor = "Black"
  BorderColor = "Black" BorderStyle = "Solid" CellSpacing = "1"
  Font-Names = "Verdana" Font-Size = "9 pt" Height = "250px" Width = "500px"
  NextPrevFormat = "ShortMonth" SelectionMode = "Day">
  <SelectedDayStyle BackColor = "DarkOrange" ForeColor = "White" />
  <DayStyle BackColor = "Orange" Font-Bold = "True" ForeColor = "White" />
  <NextPrevStyle Font-Bold = "True" Font-Size = "8 pt" ForeColor = "White" />
  <DayHeaderStyle Font-Bold = "True" Font-Size = "8 pt" ForeColor = "#333333"
    Height = "8 pt" />
  <TitleStyle BackColor = "Firebrick" BorderStyle = "None" Font-Bold = "True"
    Font-Size = "12 pt" ForeColor = "White" Height = "12 pt" />
  <OtherMonthDayStyle BackColor = "NavajoWhite" Font-Bold = "False"
    ForeColor = "DarkGray" />
</asp:Calendar>
```

This skin defines the font, colors, and styles of the Calendar control. It also sets the selection mode, the formatting of the month navigation links, and the overall size of the calendar. As a result, all you need to use this formatted calendar is the following streamlined tag:

```
<asp:Calendar ID = "Calendar1" runat = "server" />
```

■ Caution When you create skins that specify details such as sizing, be careful. When these settings are applied to a page, they could cause the layout to change, with unintended consequences. If you're in doubt, set the skin to apply only if the control specifically opts in.

Another powerful technique is to reuse images by making them part of your theme. For example, imagine you perfect an image that you want to use for OK buttons throughout your website, and another one for all Cancel buttons. The first step to implement this design is to add the images to your theme folder. For the best organization, it makes sense to create one or more subfolders just for holding images. In this example, the images are stored in a folder named ButtonImages (see Figure 12-13).

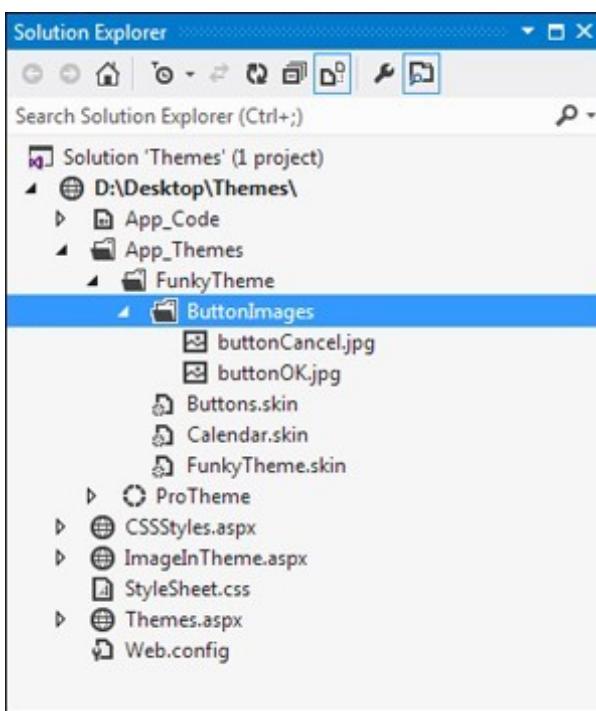


Figure 12-13. Adding images to a theme

Now, you need to create the skins that use these images. In this case, both of these tags should be named skins. That's because you're defining a specific type of standardized button that should be available to the page when needed. You *aren't* defining a default style that should apply to all buttons.

```
<asp:ImageButton runat="server" SkinID="OKButton"
ImageUrl="ButtonImages/buttonOK.jpg" />
<asp:ImageButton runat="server" SkinID="CancelButton"
ImageUrl="ButtonImages/buttonCancel.jpg" />
```

When you add a reference to an image in a skin file, always make sure the image URL is relative to the theme folder, not the folder where the page is stored. When this theme is applied to a control, ASP.NET automatically inserts the App_Themes\ThemeName portion at the beginning of the URL.

Now to apply these images, simply create an ImageButton in your web page that references the corresponding skin name:

```
<asp:ImageButton ID="ImageButton1" runat="server" SkinID="OKButton" />
<asp:ImageButton ID="ImageButton2" runat="server" SkinID="CancelButton" />
```

You can use the same technique to create skins for other controls that use images. For example, you can standardize the node pictures of a TreeView, the bullet image used for the BulletList control, or the icons used in a GridView.

appLYING theMeS DYNaMICALLY

In some cases, themes aren't used to standardize website appearance but to make that appearance configurable for each user. All you need to do to implement this design is to simply set the `Page.Theme` or `Page.StyleSheetTheme` property dynamically in your code. For example, set `Page.Theme` to the string "FunkyTheme" to apply the theme in the `FunkyTheme` directory. The only caveat is that you need to complete this step in the `Page.Init` event stage. After this point, attempting to set the property causes an exception. Similarly, you can also set the `SkinId` property of a control dynamically to attach it to a different named skin. But be careful—if a theme or skin change leads to a control specifying a skin name that doesn't exist in the current theme, an exception will be thrown.

Master Page Basics

The best websites don't look like a series of web pages; instead, they give the illusion of a continuously running application. For example, try ordering a book on Amazon. While you search, click through the links, and then head to your shopping cart, you'll always see a continuous user interface with a common header at the top, a set of navigation links on the left, and a footer at the bottom.

Creating something that polished with ASP.NET is possible, but it isn't as easy as it seems. For example, what if you want a navigation bar on every web page? Not only do you need to copy the same user interface markup to each page, but you also need to make sure it ends up in the same place. An offset of a couple of pixels will completely ruin the illusion, making it obvious that the pages aren't really integrated. And even if you copy your markup perfectly, you're still left with an extremely brittle design. If you decide to update your navigation bar or change its position later, you'll need to modify every web page to apply the same change.

So how can you deal with the complexity of different pages that need to look and act the same? One option is to subdivide the page into frames. *Frames* are an HTML feature that lets the browser show more than one web page alongside another. Unfortunately, frames have problems of their own, including that each frame is treated as a separate document and requested separately by the browser. This makes it difficult to create code that communicates between frames. A better choice is to use ASP.NET's master pages feature, which allows you to define page templates and reuse them across your website.

Note Frames are also out of favor because they limit your layout options. That's because each frame contains a separate, fixed portion of a window. When you scroll one frame, the other frames remain fixed in place. To make frames that work properly, you need to make assumptions about the target device and its screen size. Most websites (think Google, Amazon, and eBay) don't use frames.

Master pages are similar to ordinary ASP.NET pages. Like ordinary pages, master pages are text files that can contain HTML, web controls, and code. However, master pages have a different file extension (`.master` instead of `.aspx`), and they can't be viewed directly by a browser. Instead, master pages must be used by other pages, which are known as *content pages*. Essentially, the master page defines the page structure and the common ingredients. The content pages adopt this structure and just fill it with the appropriate content.

For example, if a website such as <http://www.amazon.com> were created using ASP.NET, a single master page might define the layout for the entire site. Every page would use that master page, and as a result, every page would have the same basic organization and the same title, footer, and so on. However, each page would also insert its specific information, such as product descriptions, book reviews, or search results, into this template.

A Simple Master Page and Content Page

To see how this works, it helps to create a simple example. To create a master page in Visual Studio, select Website ➤ Add New Item from the menu. Select Master Page, give it a file name (such as SiteTemplate.master, used in the next example), and click Add.

When you create a new master page in Visual Studio, you start with a blank page that includes a ContentPlaceHolder control (see Figure 12-14). The ContentPlaceHolder is the portion of the master page that a content page can change. Or, to look at it another way, everything else that's set in the master page is unchangeable in a content page. If you add a header, that header appears in every content page. If you want to give the content page the opportunity to supply content in a specific section of the page, you need to add a ContentPlaceHolder.

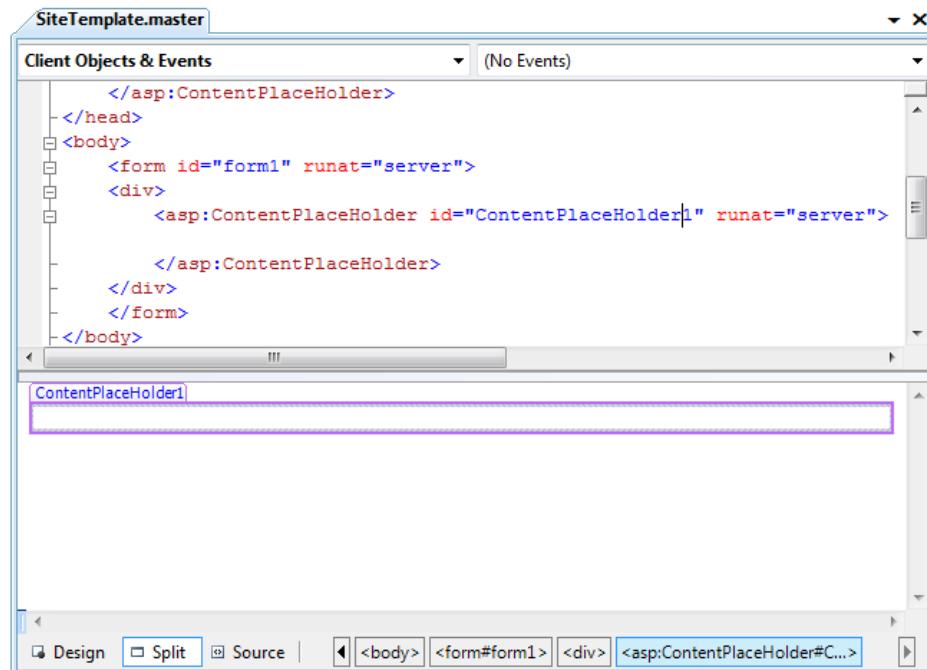


Figure 12-14. A new master page

When you first create a master page, you'll start with two ContentPlaceHolder controls. One is defined in the <head> section, which gives content pages the add page metadata, such as search keywords and style sheet links. The second, more important ContentPlaceHolder is defined in the <body> section, and represents the displayed content of the page. It appears on the page as a faintly outlined box. If you click inside it or hover over it, the name of ContentPlaceHolder appears in a tooltip, as shown in Figure 12-14.

To make this master page example more practical, try adding a header before the ContentPlaceholder (using an `` tag) and a footer after it (using some static text), as shown in Figure 12-15. You'll notice that the content area of the page looks very small, but this appearance is deceptive. The content section will expand to fit the content you place inside.

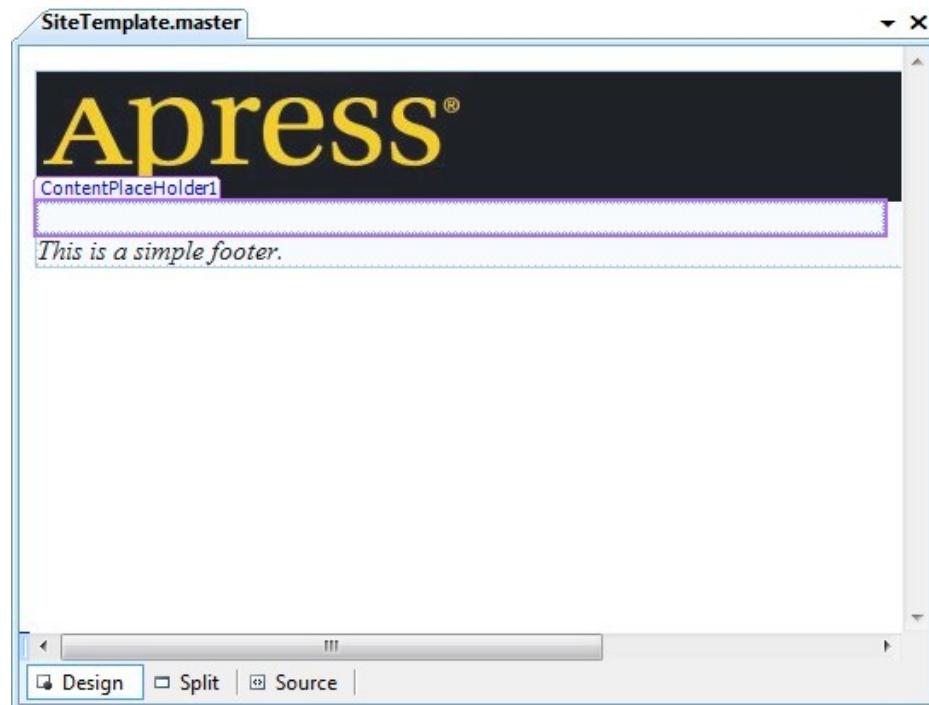


Figure 12-15. A simple master page with a header and footer

Now you're ready to create a content page based on this master page. To take this step, select Website ➤ Add New Item from the menu. Select Web Form, and choose to select a master page (see Figure 12-16). Click Add. When you're prompted to choose a master page, use the one you created with the header and footer.

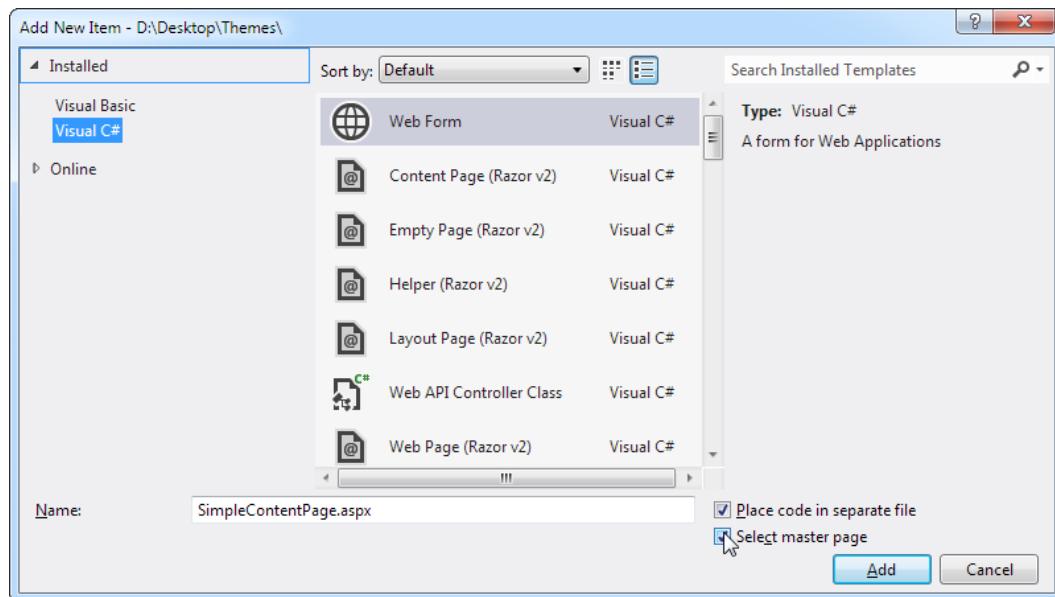


Figure 12-16. Creating a content page

Now you'll see something a little more interesting. Your content page will have all the elements of the master page, but the elements will be shaded in gray, indicating that you can't select or change them in any way. However, you can add content or drag and drop new controls into the ContentPlaceholder region to create a page like the one shown in Figure 12-17. In fact, this is the only editable portion of your page.

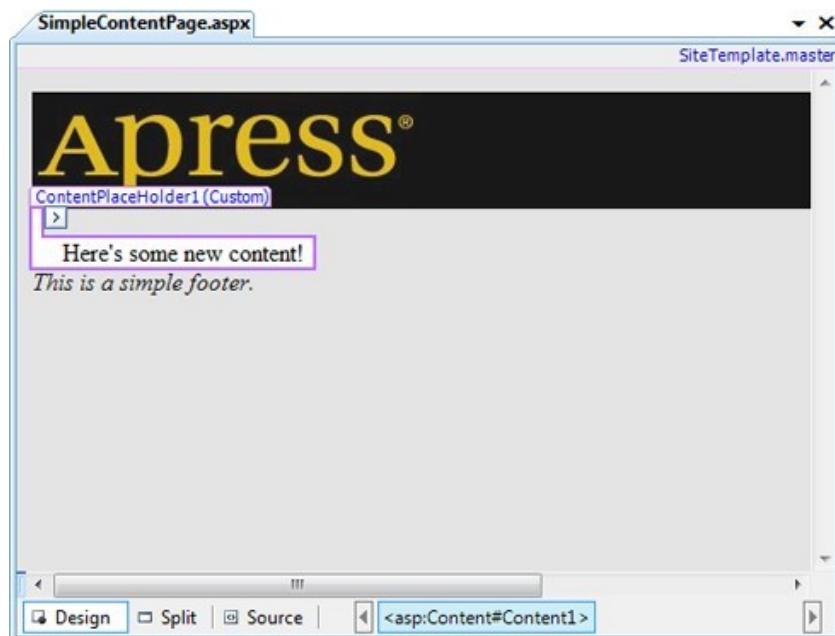


Figure 12-17. A simple content page at design time

The ContentPlaceholder section will expand or collapse to fit the content you place in it. If you've added volumes of text, the footer won't appear until the end. If you've included only a single line of text, you'll see something more compact, as in Figure 12-17. To get a clearer look at your web page, you can run it in the browser. Figure 12-18 shows the content page that's being designed in Figure 12-17.

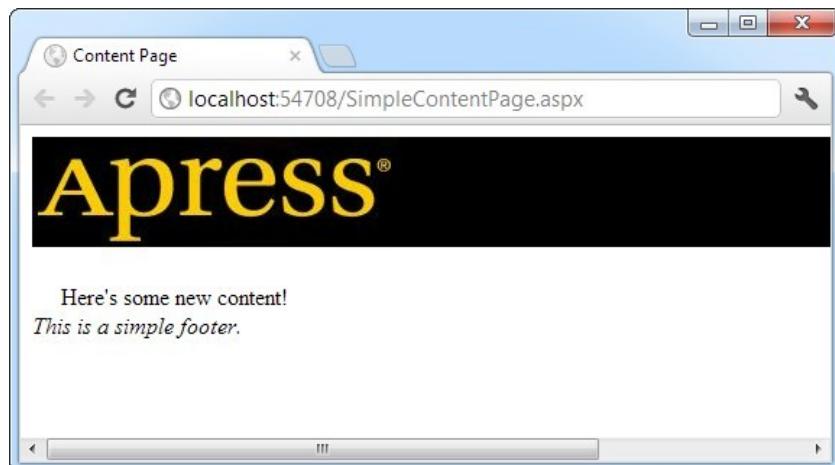


Figure 12-18. A simple content page at runtime

The real magic starts when you create multiple pages that use the same master page. Now each page will have the same header and footer, creating a seamless look across your entire website.

How Master Pages and Content Pages Are Connected

Now that you've seen a master page example, it's worth taking a look behind the scenes to see how you implement the master page.

When you create a master page, you're building something that looks much like an ordinary ASP.NET web form. The key difference is that, although web forms start with the Page directive, a master page starts with a Master directive that specifies the same information. Here's the Master directive for the simple master page shown in the previous example:

```
<%@ Master Language = "C#" AutoEventWireup = "true" CodeFile = "SiteTemplate.master.cs"
   Inherits = "SiteTemplate" %>
```

The ContentPlaceHolder is less interesting. You declare it like any ordinary control. Here's the complete code for the simple master page:

```
<%@ Master Language = "C#" AutoEventWireup = "true" CodeFile = "SiteTemplate.master.cs"
   Inherits = "SiteTemplate" %>
<html xmlns = "http://www.w3.org/1999/xhtml">
<head runat = "server">
  <title> Untitled Page</title>
</head>
<body>
  <form id = "form1" runat = "server">
    <img src = "apress.jpg" /> <br />
    <asp:ContentPlaceHolder id = "ContentPlaceHolder1" runat = "server">
      </asp:ContentPlaceHolder>
      <i> This is a simple footer.</i>
    </form>
</body>
</html>
```

Note For simplicity's sake, this example doesn't include a ContentPlaceHolder in the <head> section. Although this ContentPlaceHolder isn't required (and it's isn't used in this example), Visual Studio adds it by default for master pages.

When you create a content page, ASP.NET links your page to the master page by adding an attribute to the Page directive. This attribute, named MasterPageFile, indicates the associated master page. Here's what it looks like:

```
<%@ Page Language = "C#" MasterPageFile = "~/SiteTemplate.master"
   AutoEventWireup = "true" CodeFile = "SimpleContentPage.aspx.cs"
   Inherits = "SimpleContentPage" Title = "Untitled Page" %>
```

Notice that the MasterPageFile attribute begins with the path ~/ to specify the root website folder. Don't confuse this with a single slash (/), which simply indicates the root of the web server and is much less useful, because the root of the web server may or may not be the location of your application. Using a root relative path is the best approach because it indicates unambiguously where ASP.NET can find your master page, no matter where you deploy your application.

Note The ~/ characters create a *root-relative path*—a path that always starts from the root folder of your application. This is a special syntax understood by ASP.NET and its server controls. You can't use this syntax in ordinary HTML markup. For example, this syntax won't work in an ordinary <a> element hyperlink unless you add a runat = "server" attribute to tell ASP.NET to process it.

The Page directive has another new attribute—Title. That's because the master page, as the outermost shell of the page, always defines the <head> section of the page with a default title. Remember, your content page can't modify anything that's in the master page. However, this is an obvious shortcoming with the title information, so to circumvent it ASP.NET adds the Title attribute, which you can set to override the title specified in the master page with something more appropriate.

The rest of the content page looks a little different from an ordinary web form. That's because the content page can't define anything that's already provided in the master page, including the <head> section, the root <html> element, the <body> element, and so on. In fact, the content page can do only one thing: it can supply a Content tag that corresponds to the ContentPlaceholder in the master page. This is where you insert the content for this page. As a result, your content pages are a little bit simpler than ordinary web pages.

Here's the complete code for the simple content page, with a single line of text and two line breaks added:

```
<%@ Page Language = "C#" MasterPageFile = "~/SiteTemplate.master"
  AutoEventWireup = "true" CodeFile = "SimpleContentPage.aspx.cs"
  Inherits = "SimpleContentPage" Title = "Content Page" %>
<asp:Content ID = "Content1" ContentPlaceHolderID = "ContentPlaceHolder1"
  runat = "Server">
  <br />
  Here's some new content!
  <br />
</asp:Content>
```

For ASP.NET to process this page successfully, the ContentPlaceholderID attribute in the <Content> tag must match the ContentPlaceholder specified in the master page exactly. This is how ASP.NET knows where it should insert your content in the master page template.

Tip If a master page defines a ContentPlaceholder but your content page doesn't define a corresponding Content control, you'll see a black box in its place when you design the page in Visual Studio. To add the Content control, right-click that section of the page and choose Create Custom Content.

You should realize one important fact by looking at the content page markup. Namely, the content from the master page (the address bar and the footer) *isn't* inserted into the content file. Instead, ASP.NET grabs these details from the master page when it processes the page. This has an important effect. It means that if you want to change the header or footer that's used in all your content pages, you need to change only one file—the master page. When you make this change, it will appear in all content pages automatically. In other words, master pages don't just let you reuse standard elements; they also make it easy to update these details later.

Tip Now that you understand how to hook up master pages and child pages, you can easily take an existing page and modify it to use your master page. However, you'll need to remove some of the basic boilerplate code such as <html>, <head>, and <body>, and wrap all the content in one or more <Content> tags. Visual Studio will handle the Content control automatically except when you're creating a new content page from scratch.

A Master Page with Multiple Content Regions

Master pages aren't limited to one ContentPlaceholder. Instead, you can insert as many as you need to give the client the ability to intersperse content in various places. All you need to do is add multiple ContentPlaceholder controls and arrange them appropriately.

Figure 12-19 shows a master page that needs more careful consideration. It includes an initial ContentPlaceholder where the user can insert content, and then a shaded box (created by a <div> tag) that contains a heading (*OTHER LINKS*) and a second ContentPlaceholder. The idea here is that the page is split into two logical sections. In the content page, you won't need to worry about how to format each section or how to position the other links box. Instead, you simply supply content for each portion, and ASP.NET will insert it into the correct location in the master page.

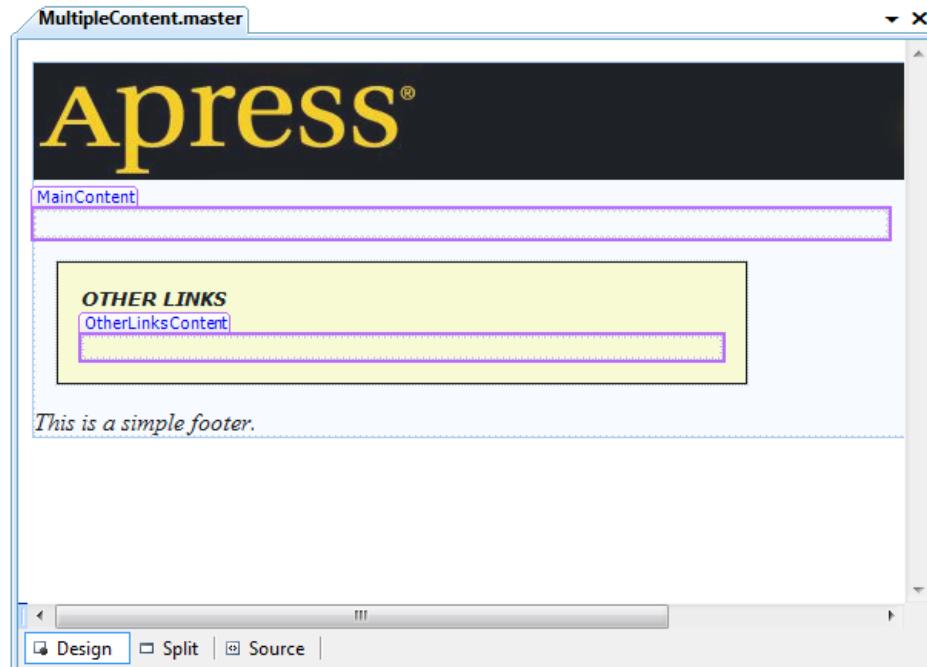


Figure 12-19. A master page with two content regions

Here's the code for the master page (with the style portion of the <div> tag omitted to save space):

```
<%@ Master Language = "C#" AutoEventWireup = "true"
   CodeFile = "MultipleContent.master.cs" Inherits = "MultipleContent" %>
<html xmlns = "http://www.w3.org/1999/xhtml">
<head runat = "server">
  <title> Untitled Page</title>
</head>
<body>
  <form id = "form1" runat = "server">
    <img src = "apress.jpg" /> <br />
    <asp:ContentPlaceHolder id = "MainContent" runat = "server">
    </asp:ContentPlaceHolder>
    <i>
      <div style = "...">
        <b> OTHER LINKS</b>
        <br />
        <asp:ContentPlaceHolder id = "OtherLinksContent" runat = "server">
        </asp:ContentPlaceHolder>
      </div>
      This is a simple footer.
    </i>
  </form>
</body>
</html>
```

Tip The most underrated part of a master page is the line break, or
 tag. If you forget to include it, you easily end up having child content run into your headings. To avoid this problem, make sure you add the correct whitespace in your master page. never rely on adding it in your content pages because content pages might not have the correct amount of space (or insert it in the correct place).

When you create a new content page based on this master page, Visual Studio will start you with one Content control for each ContentPlaceholder in the master page, making your life easy. All you need to do is insert the appropriate information. Here's a slightly shortened example, with some of the text replaced with an ellipsis (...) to save space:

```
<%@ Page Language = "C#" MasterPageFile = "~/MultipleContent.master"
   AutoEventWireup = "true" CodeFile = "MultipleContentPage.aspx.cs"
   Inherits = "MultipleContentPage" Title = "Content Page" %>
<asp:Content ID = "Content1" ContentPlaceholderID = "MainContent" runat = "Server">
  This is the generic content for this page. Here you might provide some site
  specific text ... </asp:Content>
<asp:Content ID = "Content2" ContentPlaceholderID = "OtherLinksContent"
   runat = "Server">
  Here's a <a href = "http://www.prosetech.com">link</a>. <br />
  ...
</asp:Content>
```

Figure 12-20 shows the final result. Notice how the two content sections flow into their designated locations seamlessly.

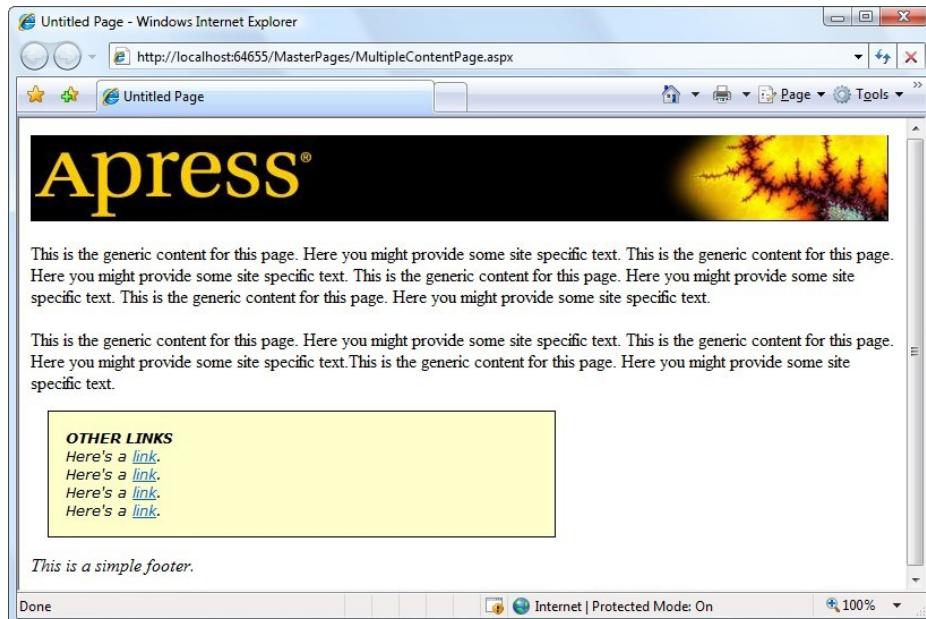


Figure 12-20. Using the multiple content master page

Another important trick is at work in this example. The master page doesn't just define the structure of the web page; it also supplies some important style characteristics (such as a default font and background color) through the <div> tag. This is another handy trick to offload the formatting work to the master page, which allows you to maintain it and modify it much more easily.

Caution If you create a master page without any ContentPlaceholder controls, content pages won't be able to supply any content at all, and they'll always show an exact copy of the master page.

Default Content

So far, you've seen master page examples with two types of content: fixed content and page-supplied content. However, in some cases your situation might not be as clear-cut. You might have some content that the content page may or may not want to replace. You can deal with this using default content.

Here's how it works: You create a master page and create a ContentPlaceholder for the content that might change. Inside that tag, you place the appropriate HTML or web controls. (You can do this by hand using the .aspx markup or just by dragging and dropping controls into the ContentPlaceholder.)

For example, here's a version of the simple header-and-footer master page shown earlier, with default content:

```
<%@ Master Language = "C#" AutoEventWireup = "true" CodeFile = "SiteTemplate.master.cs"
   Inherits = "SiteTemplate" %>
```

```
<html xmlns = "http://www.w3.org/1999/xhtml">
<head runat = "server">
    <title> Untitled Page</title>
</head>
<body>
    <form id = "form1" runat = "server">
        <img src = "apress.jpg" /> <br />
        <asp:ContentPlaceHolder id = "ContentPlaceHolder1" runat = "server">
            This is default content. <br />
        </asp:ContentPlaceHolder>
        <i> This is a simple footer.</i>
    </form>
</body>
</html>
```

So, what happens when you create a content page based on this master page? If you use Visual Studio, you won't see any change. That's because Visual Studio automatically creates a `<Content>` tag for each `ContentPlaceHolder`. When a content page includes a `<Content>` tag, it automatically overrides the default content.

However, something interesting happens if you delete the `<Content>` tag from the content page. Now when you run the page, you'll see the default content. In other words, default content appears only when the content page chooses not to specify any content for that placeholder.

You might wonder whether the content pages can use *some* of the default content or just edit it slightly. This isn't possible because the default content is stored only in the master page, not in the content page. As a result, you need to decide between using the default content as is or replacing it completely.

Tip You don't need to delete the `<Content>` tag by hand. Instead, you can use the Visual Studio smart click to select the content region in design view. Then, click the arrow that appears in the top-right corner of the content region to open the smart tag. Finally, choose Default to Master's Content (to remove the `<Content>` tag) or Create Custom Content (to add the `<Content>` tag back).

Master Pages and Relative Paths

One quirk that can catch unsuspecting developers is the way that master pages handle relative paths. If all you're using is static text, this issue won't affect you. However, if you add `` tags or any other HTML tag that points to another resource, problems can occur.

The problem shows up if you place the master page in a different directory from the content page that uses it. This is a recommended best practice for large websites. In fact, Microsoft encourages you to use a dedicated folder for storing all your master pages. However, if you're not suitably careful, this can cause problems when you use relative paths.

For example, imagine you put a master page in a subfolder named `MasterPages` and add the following `` tag to the master page:

```
<img src = "banner.jpg" />
```

Assuming the file `\MasterPages\banner.jpg` exists, this appears to work fine. The image will even appear in the Visual Studio design environment. However, if you create a content page in another subfolder, the image path is interpreted relative to that folder. If the file doesn't exist there, you'll get a broken link instead of your graphic. Even worse, you could conceivably get the wrong graphic if another image has the same file name.

This problem occurs because the `` tag is ordinary HTML. As a result, ASP.NET won't touch it. Unfortunately, when ASP.NET processes your content page, the relative path in this tag is no longer appropriate. The same problem occurs with `<a>` tags that provide relative links to other pages and with the `<link>` element that you can use to connect the master page to a style sheet.

To solve your problem, you could try to think ahead and write your URL relative to the content page where you want to use it. But this creates confusion and limits where your master page can be used. A better fix is to turn your `` tag into a server-side control, in which case ASP.NET will fix the mistake:

```
<img src = "banner.jpg" runat = "server"/>
```

This works because ASP.NET uses this information to create an `HtmlImage` server control. This object is created after the `Page` object for the master page is instantiated. At this point, ASP.NET interprets all the paths relative to the location of the master page.

And as with all server-side controls, you can further clear things up by using the `~/` characters to create a root-relative path. Here's an example that clearly points to a picture in an `Images` folder in the root web application folder:

```
<img src = "~/Images/banner.jpg" runat = "server"/>
```

Remember, the `~/` syntax is understood only by ASP.NET controls, so you can't use this trick with an `` tag that doesn't include the `runat = "server"` attribute.

Advanced Master Pages

Using what you've learned, you can create and reuse master pages across your website. However, still more tricks and techniques can help you take master pages to the next level and make them that much more practical. In the following sections, you'll look at how CSS styles can help you organize your layout and how your content pages can interact with the master page class in code.

Style-Based Layouts

For the most part, HTML uses a flow-based layout. That means that as more content is added, the page is reorganized and other content is bumped out of the way. This layout can make it difficult to get the result you want with master pages. For example, what happens if you craft the perfect layout, only to have the structure distorted by a huge block of information that's inserted into a `<Content>` tag?

Although you can't avoid this problem completely, there are two options to help you control the layout of a web page:

HTML tables: Using an HTML table in your master page, you can break a portion of your page into columns and rows. You can then add a `ContentPlaceholder` in a single cell, ensuring that the other content is aligned more or less the way you want.

CSS positioning: Using CSS styles, you can divide your content into columns or regions, each of which is held in a separate `<div>` element. You then place a single `ContentPlaceholder` in each `<div>`.

Although these two approaches are similar, the CSS standard is similar and more modern. It's the one you'll consider in the following example. (To see the equivalent example using HTML tables, refer to the downloadable samples for this chapter, which include both approaches.)

The basic concept is simple (and you've already seen it at work in some of the examples earlier in this book, like the two-panel GreetingCardMaker example from Chapter 6). First, you divide your content into multiple sections, each of which is a separately positioned column or box. In your markup, you wrap each section in

a <div> element. Finally, you apply a different style to each <div>, and you use the properties of CSS to position it appropriately.

For a good example, consider a traditional web application with a header, content panel, and two side panels. Figure 12-21 shows how this structure is broken up into separate <div> elements, each of which uses a separate style in an external style sheet.

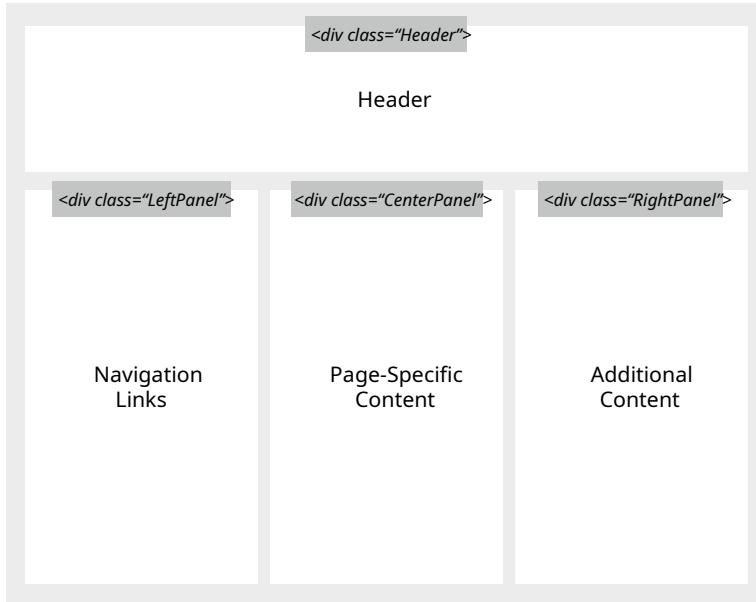


Figure 12-21. A style-based layout

The trick to creating this sort of multiple-column layout is to use absolute positioning to place the columns. The first step to implanting this design is to create four styles, named Header, LeftPanel, RightPanel, and CenterPanel (all of which you should place in an external style sheet). Here's what you need for the Header style:

```
.Header
{
    position: absolute;
    top: 10px;
    left: 10px;
    height: 60px;
    text-align: center;
}
```

This creates an absolutely positioned box that sits close to the top of the web page, taking the full available width but limiting itself to just 60 pixels of height (any excess content is cut off).

The LeftPanel style is more interesting:

```
.LeftPanel
{
    position: absolute;
    top: 70px;
    left: 10px;
    width: 160px;
}
```

Here, the style rule creates a left column that's 160 pixels wide. The column is placed 70 pixels from the top of the page and 10 pixels from left edge. Because these coordinates are relative to the side of the browser window, it really doesn't matter how big or small that window is.

A similar style rule places the rightmost column:

```
.RightPanel
{
    position: absolute;
    top: 70px;
    right: 10px;
    width: 160px;
}
```

Like the left column, the right panel is 160 pixels. The difference is that it's positioned away from the right edge of the browser window (because the style sets the *right* property, not the *left* property).

The final style creates the content region. This style can't use a fixed size because the exact width depends on how much space is left over after the two panels are inserted on the sides. This, in turn, depends on the current width of the browser window. Fortunately, there's an easy way to deal with this unknown: you simply set the left and right margins large enough to fit the left and right columns, and your content will flow neatly into the available space.

For example, the left panel is 160 pixels wide and is placed 10 pixels from the left edge of the browser window, which means you need a left margin of 170 pixels, plus a few pixels of extra breathing space to prevent content or borders from touching. Here's a suitable style rule:

```
.CenterPanel
{
    position: absolute;
    top: 70px;
    margin-left: 175px;
    margin-right: 180px;
}
```

Now, you can use the styles in the master web page:

```
<div class = "Header">
</div>

<div class = "LeftPanel">
</div>

<div class = "CenterPanel">
</div>
```

```
<div class = "RightPanel">
</div>
```

It's worth emphasizing that when you use this technique, the order of the `<div>` sections doesn't matter because each one is placed at a precise region. (The only exception is if the sections overlap, in which case the later sections can overwrite the earlier ones.) And because these rules are in an external style sheet, it's easy to use them in multiple pages or to tweak them later without editing your page.

The final step is to insert and fixed content and the `ContentPlaceHolder` elements. Here's a master page that does exactly that:

```
<html xmlns = "http://www.w3.org/1999/xhtml">
<head runat = "server">
    <title> </title>
    <link href = "LayoutStyles.css" rel = "stylesheet" type = "text/css" />
</head>
<body>
    <form id = "form1" runat = "server">

        <div class = "Header">
            <h1> My Header</h1>
        </div>

        <div class = "LeftPanel">
            <asp:TreeView ID = "TreeView1" runat = "server" Width = "150px">
                <Nodes>
                    <asp:TreeNode Text = "Root" Value = "New Node">
                        <asp:TreeNode Text = "Page 1" Value = "Page 1"> </asp:TreeNode>
                        <asp:TreeNode Text = "Page 2" Value = "Page 2"> </asp:TreeNode>
                    </asp:TreeNode>
                </Nodes>
            </asp:TreeView>
        </div>

        <div class = "CenterPanel">
            <asp:ContentPlaceHolder id = "MainContent" runat = "server">
            </asp:ContentPlaceHolder>
        </div>

        <div class = "RightPanel">
            <asp:ContentPlaceHolder id = "AdditionalContent" runat = "server">
            </asp:ContentPlaceHolder>
        </div>

    </form>
</body>
</html>
```

This master page uses fixed text in the header and a fixed `TreeView` for navigation in the left panel. (In this example, the `TreeView` also has a few dummy nodes added, just so that it appears in the page. When using this design in a real website, you'd bind the `TreeView` to a site map, as described in Chapter 13.) This master page uses a `ContentPlaceHolder` in the middle content region (for the main page content) and another one in the right-side panel (for additional page content, such as related topic links, author or summary details, an advertisement, and so on).

Figure 12-22 shows the resulting master page and a content page that uses the master page (both in Visual Studio). You'll notice that the different <div> elements have been given different background colors (using the corresponding style), which makes it easier to distinguish the separate regions when you're trying this example.

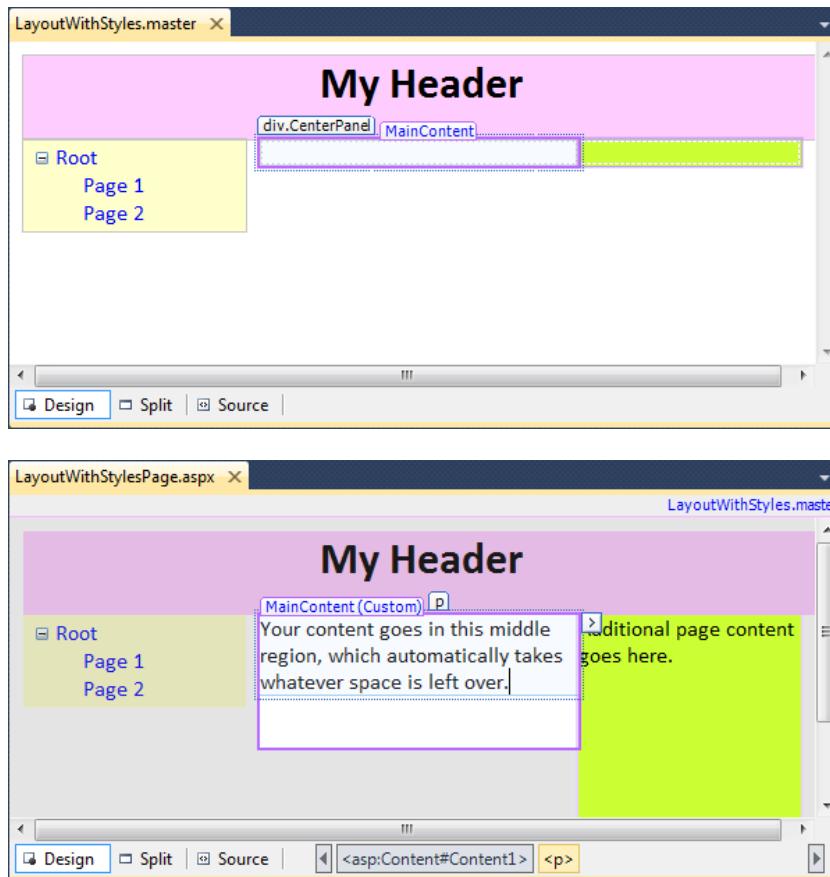


Figure 12-22. A master page (top) and a content page (bottom) using style-based layout

To convert this example into something more practical, just replace the hard-coded text and TreeView nodes in the master page with the actual header and navigation controls you really want. All the child pages will acquire these features automatically. This is the first step toward defining a practical structure for your entire website. You'll also want to tweak the styles to have the formatting you want, and you'll probably want to add a bit of space around the content in each region using the padding properties.

Tip To get some style sheet examples for basic multicolumn layouts (www.cssplay.com/web/layouts.html). For more ambitious examples of styles at work (www.cssgarden.com). There you'll see a richly formatted page that can be completely reformatted and rearranged just by switching its style sheet. All the site includes more 200 sample style sheets that you can download and try yourself.

NeStING MaSter paGeS

you can nest master pages so that one master page uses another master page. This is not done too often, but it could allow you to standardize your website to different degrees. For example, you might have different sections of your website. Each section might require its own navigation controls; however, both sections need the same header. In this case, you could create a top-level master page that adds the header. Then you would create a second master page that uses the first master page (through the masterPageFile attribute). This second master page would get the header and could add the navigation controls. You would create multiple versions of this second master page, one for each section of your website. Finally, your content pages would use one of the two second-level master pages to standardize their layout.

Be careful when implementing this approach—although it sounds like a nifty way to make a modular website, it can tie you down more than you realize. For example, you'll need to rework your master page hierarchy if you decide later that the two website sections need similar but slightly different headers. For these reasons, it's usually better to use only one level of master pages and copy the few elements that are in common. In most cases, you won't be creating many master pages, so this won't add a significant amount of duplication.

Code in a Master Page

In all the examples in this chapter, master pages have provided static layout. However, just like a web page, master pages also include a code portion that can respond to events in the page life cycle or the constituent controls. For example, you could respond to the `Page.Load` event to initialize a master page using code, or you could handle clicks in a set of navigation controls to direct a user to the right page.

Interacting with a Master Page Programmatically

A master control isn't limited to event handling code. It can also provide methods that the content page can trigger as needed or provide properties that the content page can set according to its needs. This allows the content page to interact with the master page.

For example, imagine you want to give the user the ability to collapse the cell with the navigation controls to have more room to see the page content. You don't want to implement this feature in the master page because you want it to be available only on certain pages. However, the content page obviously can't implement this feature on its own because it involves modifying a fixed portion of the master page. The solution is to create a way for the content page to interact with the master page so it can politely ask the master page to collapse or hide the navigation controls as needed.

One good way to implement this design is by adding a new property named `ShowNavigationControls` to the master page class. This property, when set to false, could then automatically hide the navigation controls. Here's the property you need to add to the master page class:

```
public bool ShowNavigationControls
{
    get
    {
        return TreeView1.Visible;
    }
    set
    {
        TreeView1.Visible = value;
    }
}
```

You should notice a few important facts about this property. First, it's public so that other classes (and therefore other pages) can access it. Second, it just wraps the Visible property in the TreeView control on the master page. Whatever value is passed to ShowNavigationControls is simply applied to TreeView.Visible. This is useful because ordinarily the TreeView.Visible property isn't directly accessible to the content page.

To access this page, the content page uses the built-in Page.Master property. This page always returns the linked object for the master page. However, you can't access the ShowNavigationControls property directly as Page.Master.ShowNavigationControls, because the Page.Master property uses the base MasterPage class and doesn't know anything about the properties you've added to your derived master page class. To get access to the custom members you've added (like ShowNavigationControls), you need to cast the Page.Master object to the appropriate type.

Here's the button handling code for a content page that hides or shows the navigation controls depending on whether a Hide or Show button is clicked. In this example, the master page class is named TableMaster.

```
protected void cmdHide_Click(object sender, EventArgs e)
{
    TableMaster master = (TableMaster)this.Master;
    master.ShowNavigationControls = false;
}

protected void cmdShow_Click(object sender, EventArgs e)
{
    TableMaster master = (TableMaster)this.Master;
    master.ShowNavigationControls = true;
}
```

Figure 12-23 shows this content page in action.

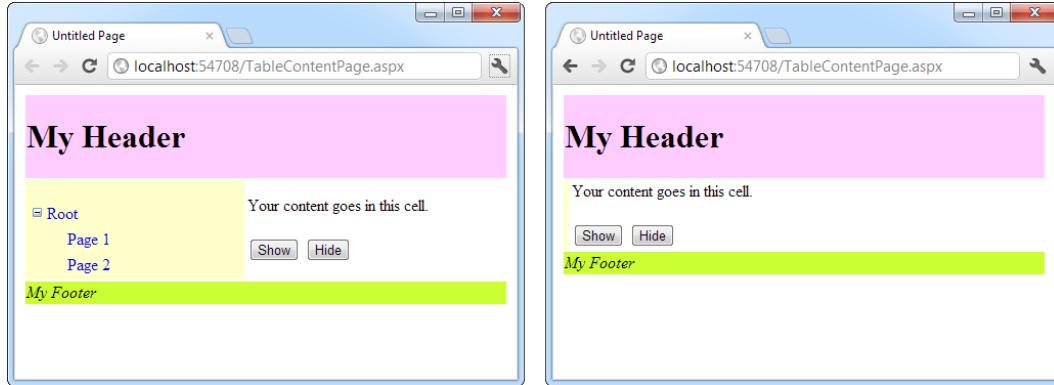


Figure 12-23. A content page that interacts with its master page

Note that when you navigate from one page to another, all the web page objects are re-created. Even if you move to another content page that uses the same master page, ASP.NET creates a different instance of the master page object. As a result, the TreeView.Visible property of the navigation controls is reset to its default value (true) every time the user navigates to a new page. If this isn't the effect you want, you need to store the setting somewhere else (such as in a cookie or in session state). Then you can write code in the master page that always checks the last saved value. Chapter 8 has more information about the ways you can store information in an ASP.NET website.

The Last Word

Building a professional web application involves much more than designing individual web pages. You also need the tools to integrate your web pages in a complete, unified website. In this chapter, you considered three ways to do exactly that. First, you considered CSS, which lets you apply consistent formatting to HTML elements and web controls alike. Then, you considered the ASP.NET themes feature, which lets you effortlessly apply a group of property settings to a control. Finally, you learned to use master pages, which allow you to standardize the layout of your website. All these features make it easy to bring your pages together into a well-integrated, consistent web application.



ADO.NET Fundamentals

At the beginning of this book, you learned that ASP.NET is just one component in Microsoft's ambitious .NET platform. As you know, .NET also includes modern languages and a toolkit of classes that allows you to do everything from handling errors to analyzing XML documents. In this chapter, you'll explore another one of the many features in the .NET Framework: the ADO.NET data access model.

Quite simply, ADO.NET is the technology that .NET applications use to interact with a database. In this chapter, you'll learn about ADO.NET and the family of objects that provides its functionality. You'll also learn how to put these objects to work by creating simple pages that retrieve and update database records. However, you won't learn about one of the most interesting ways to access a database—using a code-generation and data-modeling tool called LINQ to Entities. Although LINQ to Entities is a powerful and practical way to generate a data model for your database, it may be overkill for your application, it may be unnecessarily complex, or it may not give you all the control you need (for example, if you want to perform unusual data tasks or implement elaborate performance-optimizing techniques). For these reasons, every ASP.NET developer should start by learning the ADO.NET fundamentals that are covered in this chapter.

Note The LINQ to Entities feature is a *higher-level* model. That means it uses the ADO.NET classes you're about in this chapter to do its dirty work. After you've mastered the essentials of ADO.NET, you'll be ready for LINQ to Entities in Chapter 24.

Understanding Databases

Almost every piece of software ever written works with data. In fact, a typical web application is often just a thin user interface shell on top of sophisticated data-driven code that reads and writes information from a database. Often website users aren't aware (or don't care) that the displayed information originates from a database. They just want to be able to search your product catalog, place an order, or check their payment records.

The most common way to manage data is to use a database. Database technology is particularly useful for business software, which typically requires sets of related information. For example, a typical database for a sales program consists of a list of customers, a list of products, and a list of sales that draws on information from the other two tables. This type of information is best described by using a *relational model*, which is the philosophy that underlies all modern database products, including SQL Server, Oracle, and even Microsoft Access.

As you probably know, a relational model breaks information down to its smallest and most concise units. For example, a sales record doesn't store all the information about the products that were sold. Instead, it stores just a product ID that refers to a full record in a product table, as shown in Figure 14-1.

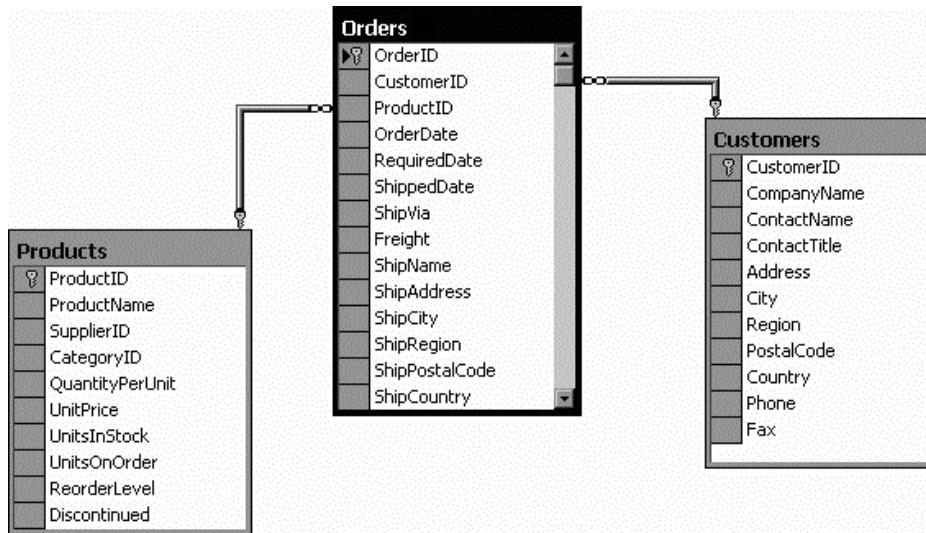


Figure 14-1. Basic table relationships

Although it's technically possible to organize data into tables and store it on the hard drive in one or more files (perhaps using a standard such as XML), this approach wouldn't be very flexible. Instead, a web application needs a full *relational database management system* (RDBMS), such as SQL Server. The RDBMS handles the data infrastructure, ensuring optimum performance and reliability. For example, the RDBMS takes the responsibility of providing data to multiple users simultaneously, disallowing invalid data, and using transactions to commit groups of actions at once.

In most ASP.NET applications, you'll need to use a database for some tasks. Here are some basic examples of data at work in a web application:

- E-commerce sites (for example, Amazon.com) use detailed databases to store product catalogs. They also track orders, customers, shipment records, and inventory information in a huge arrangement of related tables.
- Search engines (for example, Google) use databases to store indexes of page URLs, links, and keywords.
- Knowledge bases (for example, Microsoft Support) use less-structured databases that store vast quantities of information or links to various documents and resources.
- Media sites (for example, The New York Times) store their articles in databases.

You probably won't have any trouble thinking about where you need to use database technology in an ASP.NET application. What web application couldn't benefit from a guest book that records user comments or a simple e-mail address submission form that uses a back-end database to store a list of potential customers or contacts? This is where ADO.NET comes into the picture. ADO.NET is a technology designed to let an ASP.NET program (or any other .NET program, for that matter) access data.

Tip If you're a complete database novice, you can get up to speed on essential database concepts by the video tutorials at www.asp.net/sql-server/videos. There you'll find more than nine hours of instruction that describes how to use SQL Server Express. (The content was originally prepared with SQL Server Express 2008, but remains relevant for SQL Server 2012.) The tutorials move from absolute basics—covering topics such as data types and table relationships—to more-advanced subject matter such as full-text search, reporting, and network security.

Configuring Your Database

Before you can run any data access code, you need a database server to take your command. Although there are dozens of good options, all of which work equally well with ADO.NET (and require essentially the same code), a significant majority of ASP.NET applications use Microsoft SQL Server.

This chapter includes code that works with SQL Server 7 or later, although you can easily adapt the code to work with other database products. If you don't have a full version of SQL Server, there's no need to worry—you can use the free SQL Server Express (as described in the next section). It includes all the database features you need to develop and test a web application.

Note This chapter (and the following two chapters) use examples drawn from the pubs and Northwind databases. These databases aren't preinstalled in modern versions of SQL Server. However, you can easily install them using the scripts provided with the online samples. See the `readme.txt` file for full instructions, or refer to the "Using the sqlcmd Command-Line Tool" section later in this chapter.

Using SQL Server Express

If you don't have a test database server handy, you may want to use SQL Server 2012 Express Edition. It's a scaled-down version of SQL Server that's free to distribute. SQL Server Express has certain limitations—for example, it can use only one CPU, four internal CPU cores, and a maximum of 1GB of RAM. Also, SQL Server Express databases can't be larger than 10GB. However, it's still remarkably powerful and suitable for many midscale websites. Even better, you can easily upgrade from SQL Server Express to a paid version of SQL Server if you need more features later.

There are actually two versions of SQL Server 2012 Express, which makes life slightly confusing. You can download the standalone edition, with or without database tools, from www.microsoft.com/express/sql. This is the version you want if you're installing it on a real web server (and you don't have the full version of SQL Server). The second version is the awkwardly named SQL Server 2012 Express LocalDB, which is included with all versions of Visual Studio. It has almost exactly the same functionality, but it's intended to be a testing tool for development purposes only (and it doesn't include any extra tools).

In this chapter, we refer to both editions as SQL Server Express, unless we need to highlight a difference between the two.

For a comparison between SQL Server Express and other editions of SQL Server, refer to www.microsoft.com/sqlserver/en/us/editions.aspx. SQL Server Express is included with the Visual Studio installation, but if you need to download it separately or you want to download the free graphical management tools that work with it, go to www.microsoft.com/express/database.

Browsing and Modifying Databases in Visual Studio

As an ASP.NET developer, you may have the responsibility of creating the database required for a web application. Alternatively, the database may already exist, or it may be the responsibility of a dedicated database administrator. If you're using a full version of SQL Server, you'll probably use a graphical tool such as SQL Server Management Studio to create and manage your databases.

Tip SQL server Express doesn't include SQL server management studio in the download that you use it. However, you can download it separately from www.microsoft.com/express/databases. Click the Download SQL server 2012 Express button to show a list of downloadable packages. Then click the Download button next to SQL server management studio Express (Tools Only).

If you don't have a suitable tool for managing your database, or you don't want to leave the comfort of Visual Studio, you can perform many of the same tasks by using Visual Studio's Server Explorer window. (Confusingly enough, the Server Explorer window is called the Database Explorer window in Visual Studio Express for Web.)

You may see a tab for the Server Explorer on the right side of the Visual Studio window, grouped with the Toolbox and collapsed. If you do, click the tab to expand it. If not, choose **View > Server Explorer** to show it (or **View > Database Explorer** in Visual Studio Express for Web).

Using the Data Connections node in the Server Explorer, you can connect to existing databases or create new ones. Assuming you've installed the pubs database (see the *readme.txt* file for instructions), you can create a connection to it by following these steps:

1. Right-click the Data Connections node and choose Add Connection.
2. When the Choose Data Source window appears, select Microsoft SQL Server and then click Continue.
3. If you're using a full version of SQL Server, enter **localhost** as your server name. This indicates that the database server is the default instance on the local computer. (Replace this with the name of a remote computer if needed.) SQL Server Express is a bit different. If you're using SQL Server Express LocalDB (the version that's included with Visual Studio), you need to enter **(localdb)\v11.0** instead of **localhost**, as shown in Figure 14-2. If you've downloaded the full edition of SQL Server Express, you need to enter **localhost\SQLEXPRESS** instead.

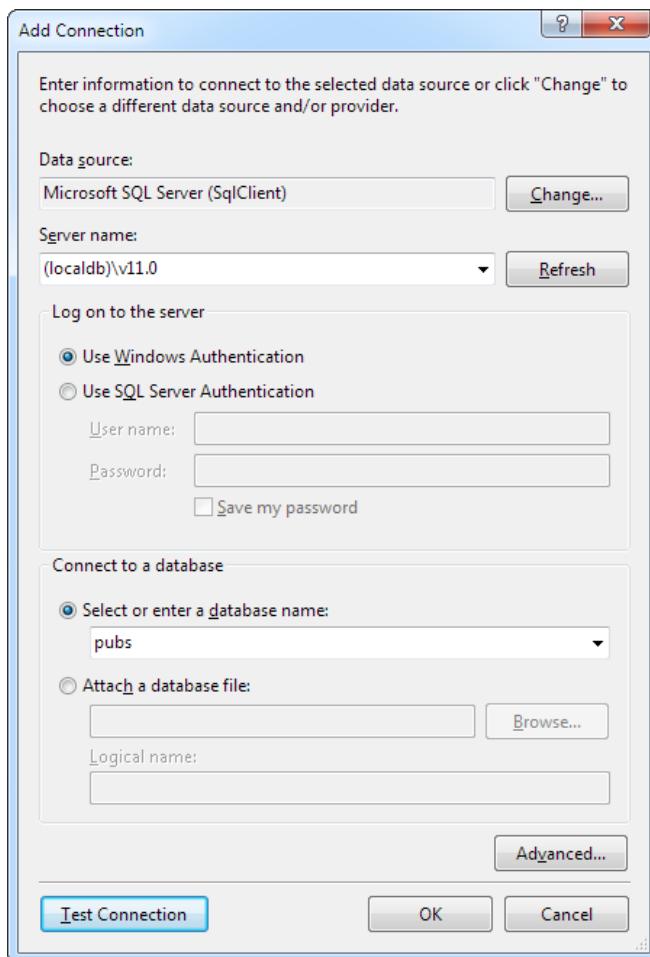


Figure 14-2. Creating a connection in Visual Studio

4. Click the Test Connection button to verify that this is the location of your database. If you haven't installed a database product yet, this step will fail. Otherwise, you'll know that your database server is installed and running.
5. In the Select or Enter a Database Name list, choose the pubs database. (In order for this to work, the pubs database must already be installed. You can install it by using the database script that's included with the sample code, as explained in the following section.) If you want to see more than one database in Visual Studio, you'll need to add more than one data connection.

Tip Alternatively, you can choose to create a new database by right-clicking the Data Connections node and choosing Create New SQL Server Database.

- Click OK. The database connection appears in the Server Explorer window. You can now explore its groups to see and edit tables, stored procedures, and more. For example, if you right-click a table and choose Show Table Data, you'll see a grid of records that you can browse and edit, as shown in Figure 14-3.

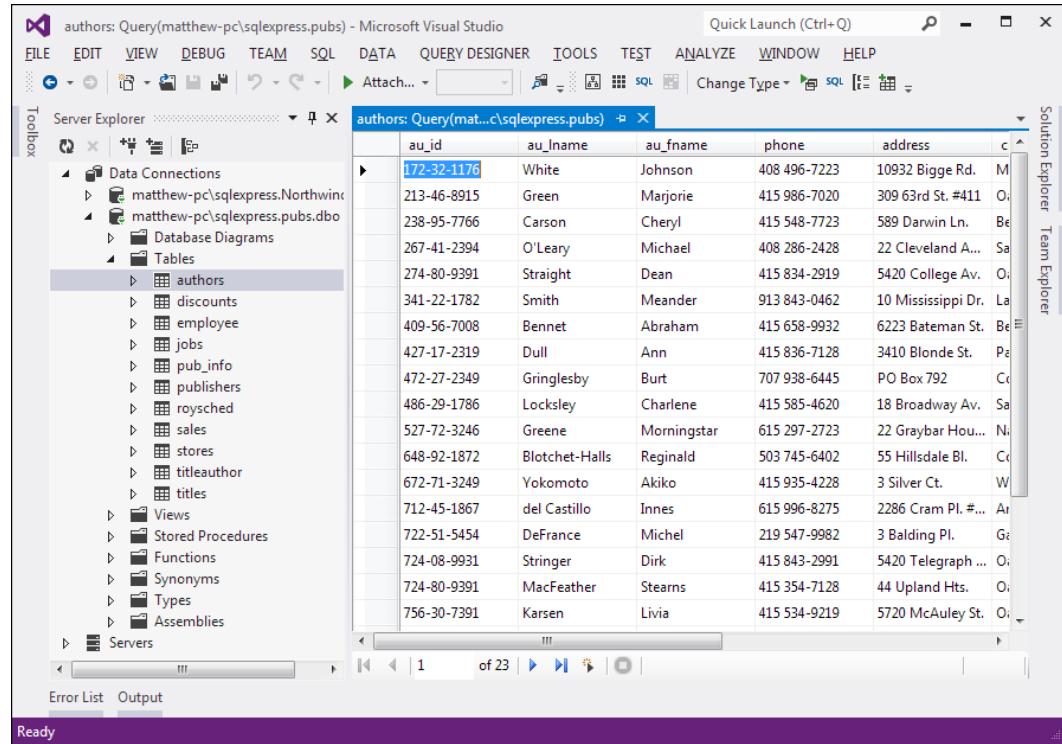


Figure 14-3. Editing table data in Visual Studio

Tip The server Explorer window is particularly handy if you're using SQL Server Express, which gives you the ability to place databases directly in the App_Data folder of your web application (instead of placing all databases in a separate, dedicated location). If Visual Studio finds a database in the App_Data folder, it automatically adds a connection for it to the Data Connections group. To learn more about this feature, check out the "Adding Instance Connections" section later in this chapter.

Using the sqlcmd Command-Line Tool

SQL Server includes a handy command-line tool named *sqlcmd.exe* that you can use to perform database tasks from a Windows command prompt. Compared to a management tool such as SQL Server Management Studio, *sqlcmd* doesn't offer many frills. It's just a quick-and-dirty way to perform a database task. Often *sqlcmd* is used in a batch file—for example, to create database tables as part of an automated setup process.

The *sqlcmd* tool is found in the directory c:\Program Files\Microsoft SQL Server\110\Tools\Binn. To run it, you need to open a Command Prompt window in this location. (One easy way to do that is to browse to the version folder in Windows Explorer, hold down Shift, and right-click the Binn folder. Then, choose "Open command menu window here" from the menu.) Once you're in the right folder, you can use *sqlcmd* to connect to your database and execute scripts.

When running *sqlcmd*, it's up to you to supply the right parameters. To see all the possible parameters, type this command:

```
sqlcmd -?
```

Two commonly used *sqlcmd* parameters are *-S* (which specifies the location of your database server) and *-i* (which supplies a script file with SQL commands that you want to run). For example, the downloadable code samples include a file named InstPubs.sql that contains the commands you need to create the pubs database and fill it with sample data. If you're using SQL Server Express LocalDB, and you've installed the code in a folder named c:\Beginning ASP.NET, you can run the InstPubs.sql script like this:

```
sqlcmd -S (localdb)\v11.0 -i "c:\Beginning ASP.NET\InstPubs.sql"
```

If you're using a full version of SQL Server on the local computer, you don't need to supply the server name at all:

```
sqlcmd -i "c:\Beginning ASP.NET\InstPubs.sql"
```

And if your database is on another computer, you need to supply that computer's name with the *-S* parameter (or just run *sqlcmd* on that computer).

Note The parameters you use with *sqlcmd* are case sensitive. For example, if you use *-s* instead of *-S*, receive an obscure error message informing you that *sqlcmd* couldn't log in.

Figure 14-4 shows the feedback you'll get when you run InstPubs.sql with *sqlcmd*.

```
D:\Code\Beginning ASP.NET>sqlcmd -S localhost\SQLExpress -i InstPubs.sql
Changed database context to 'master'.
Beginning InstPubs.SQL at 25 Jun 2012 23:53:51:137 ....
Creating pubs database....
Changed database context to 'pubs'.
Now at the create table section ....
Now at the create trigger section ....
Now at the inserts to authors ....
Now at the inserts to publishers ....
Now at the inserts to pub_info ....
Now at the inserts to titles ....
Now at the inserts to titleauthor ....
Now at the inserts to stores ....
Now at the inserts to sales ....
Now at the inserts to roysched ....
Now at the inserts to discounts ....
Now at the inserts to jobs ....
Now at the inserts to employee ....
Now at the create index section ....
Now at the create view section ....
Now at the create procedure section ....
Changed database context to 'master'.
Ending InstPubs.SQL at 25 Jun 2012 23:53:52:017 ....
D:\Code\Beginning ASP.NET>
```

Figure 14-4. Running an SQL script with `sqlcmd.exe`

In this book, you'll occasionally see instructions about using `sqlcmd` to perform some sort of database configuration. However, you can usually achieve the same result (with a bit more clicking) by using the graphical interface in a tool such as SQL Server Management Studio. For example, to install a database by running an SQL script, you simply need to start SQL Server Management Studio, open the SQL file (using `File > Open > File` command), and then run it (using the `Query > Execute` command).

Understanding SQL Basics

When you interact with a data source through ADO.NET, you use Structured Query Language (SQL) to retrieve, modify, and update information. In some cases, ADO.NET will hide some of the details for you or even generate required SQL statements automatically. However, to design an efficient database application with a minimal amount of frustration, you need to understand the basic concepts of SQL.

SQL is a standard data access language used to interact with relational databases. Different databases differ in their support of SQL or add other features, but the core commands used to select, add, and modify data are common. In a database product such as SQL Server, it's possible to use SQL to create fairly sophisticated SQL scripts for stored procedures and triggers (although they have little of the power of a full object-oriented programming language). When working with ADO.NET, however, you'll probably use only the following standard types of SQL statements:

- A Select statement retrieves records.
- An Update statement modifies existing records.
- An Insert statement adds a new record.
- A Delete statement deletes existing records.

If you already have a good understanding of SQL, you can skip the next few sections. Otherwise, read on for a quick tour of SQL fundamentals.

Tip To learn more about SQL, use one of the SQL tutorials available on the Internet, such as the one at www.w3schools.com/sql. If you're working with SQL server, you can use its thorough Books Online help to become a database guru.

Running Queries in Visual Studio

If you've never used SQL before, you may want to play around with it and create some sample queries before you start using it in an ASP.NET site. Most database products provide some sort of tool for testing queries. If you're using a full version of SQL Server, you can try SQL Server Management Studio. If you don't want to use an extra tool, you can run your queries by using the Server Explorer window described earlier. Just follow these steps in Visual Studio:

1. Right-click your connection and choose New Query.
2. Choose the table (or tables) you want to use in your query from the Add Table dialog box (as shown in Figure 14-5). Click Add and then click Close.

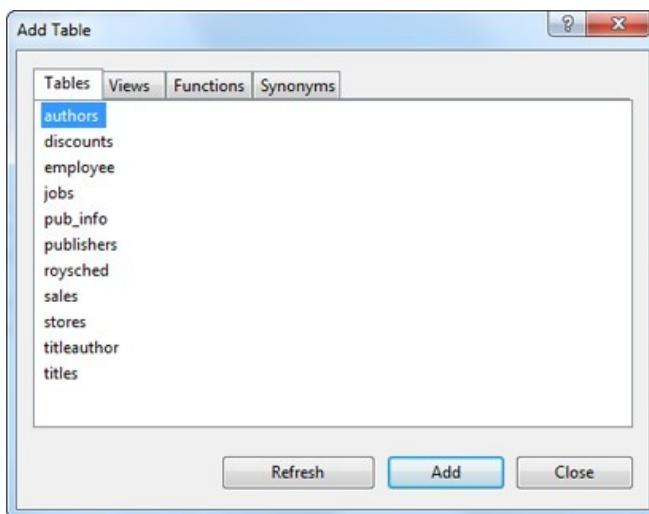


Figure 14-5. Adding tables to a query

3. You'll now see a handy query-building window. You can create your query by adding check marks next to the fields you want, or you can edit the SQL by hand in the lower portion of the window. Best of all, if you edit the SQL directly, you can type in anything—you don't need to stick to the tables you selected in step 2, and you don't need to restrict yourself to Select statements.
4. When you're ready to run the query, choose **Query Designer** → **Execute SQL** from the menu. Assuming your query doesn't have any errors, you'll get one of two results. If you're selecting records, the results will appear at the bottom of the window. If you're deleting or updating records, a message box will appear, informing you of the number of records affected (see Figure 14-6).

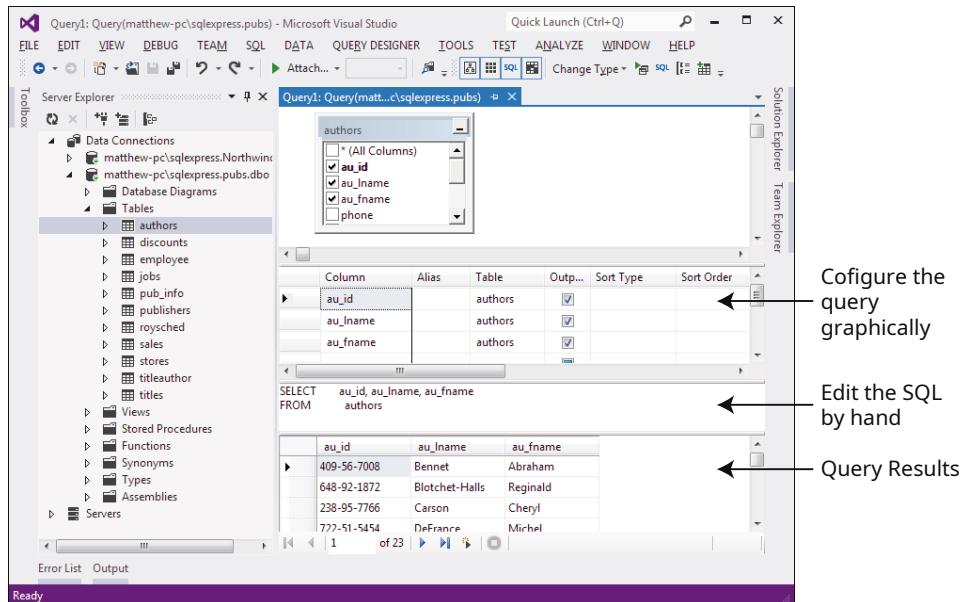


Figure 14-6. Executing a query

Tip When programming with ADO.NET, it always helps to know your database. If you have information about the data types it uses, the stored procedures it provides, and the user account you need to use, you can work more quickly and with less chance of error.

Using the Select Statement

To retrieve one or more rows of data, you use a `Select` statement. A basic `Select` statement has the following structure:

```
SELECT [columns]
FROM [tables]
WHERE [search_condition]
ORDER BY [order_expression ASC | DESC]
```

This format really just scratches the surface of SQL. If you want, you can create more-sophisticated queries that use subgrouping, averaging and totaling, and other options (such as setting a maximum number of returned rows). By performing this work in a query (instead of in your application), you can often create far more efficient applications.

The next few sections present sample `Select` statements. After each example, a series of bulleted points breaks down the SQL to explain how each part of it works.

A Sample Select Statement

The following is a typical (and rather inefficient) Select statement for the pubs database. It works with the Authors table, which contains a list of authors:

```
SELECT * FROM Authors
```

- The asterisk (*) retrieves all the columns in the table. This isn't the best approach for a large table if you don't need all the information. It increases the amount of data that has to be transferred and can slow down your server.
- The From clause identifies that the Authors table is being used for this statement.
- The statement doesn't have a Where clause. This means all the records will be retrieved from the database, regardless of whether it has 10 or 10 million records. This is a poor design practice, because it often leads to applications that appear to work fine when they're first deployed but gradually slow down as the database grows. In general, you should always include a Where clause to limit the possible number of rows (unless you absolutely need them all). Often queries are limited by a date field (for example, including all orders that were placed in the last three months).
- The statement doesn't have an Order By clause. This is a perfectly acceptable approach, especially if order doesn't matter or you plan to sort the data on your own by using the tools provided in ADO.NET.

An Improved Select Statement

Here's another example that retrieves a list of author names:

```
SELECT au_lname, au_fname FROM Authors WHERE State='CA' ORDER BY au_lname ASC
```

- Only two columns are retrieved (au_lname and au_fname). They correspond to the first and last names of the author.
- A Where clause restricts results to those authors who live in the specified state (California). Note that the Where clause requires apostrophes around the value you want to match, because it's a text value.
- An Order By clause sorts the information alphabetically by the author's last name. The ASC part (for ascending) is optional, because that's the default sort order.

An Alternative Select Statement

Here's one last example:

```
SELECT TOP 100 * FROM Sales ORDER BY ord_date DESC
```

This example uses the Top clause instead of a Where statement. The database rows will be sorted in descending order by order date, and the first 100 matching results will be retrieved. In this case, it's the 100 most recent orders. You could also use this type of statement to find the most expensive items you sell or the best-performing employees.

The Where Clause

In many respects, the Where clause is the most important part of the Select statement. You can find records that match several conditions by using the And keyword, and you can find records that match any one of a series of conditions by using the Or keyword. You can also specify greater-than and less-than comparisons by using the greater-than (>) and less-than (<) operators.

The following is an example with a different table and a more sophisticated Where statement:

```
SELECT * FROM Sales WHERE ord_date < '2000/01/01' AND ord_date > '1987/01/01'
```

This example uses the international date format to compare date values. Although SQL Server supports many date formats, yyyy/mm/dd is recommended to prevent ambiguity.

If you were using Microsoft Access, you would need to use the US date format, mm/dd/yyyy, and replace the apostrophes around the date with the number (#) symbol.

String Matching with the Like Operator

The Like operator allows you to perform partial string matching to filter records in order to find a particular field that starts with, ends with, or contains a certain set of characters. For example, if you want to see all store names that start with *B*, you could use the following statement:

```
SELECT * FROM Stores WHERE stor_name LIKE 'B%'
```

To see a list of all stores *ending* with *S*, you would put the percent sign *before* the *S*, like this:

```
SELECT * FROM Stores WHERE stor_name LIKE '%S'
```

The third way to use the Like operator is to return any records that contain a certain character or sequence of characters. For example, suppose you want to see all stores that have the word *book* somewhere in the name. In this case, you could use an SQL statement like this:

```
SELECT * FROM Stores WHERE stor_name LIKE '%book%'
```

By default, SQL is not case sensitive, so this syntax finds instances of *BOOK*, *book*, or any variation of mixed case.

Finally, you can indicate one of a set of characters, rather than just any character, by listing the allowed characters within square brackets. Here's an example:

```
SELECT * FROM Stores WHERE stor_name LIKE '[abcd]%'
```

This SQL statement will return stores with names starting with *A*, *B*, *C*, or *D*.

Aggregate Queries

The SQL language also defines special *aggregate functions*. Aggregate functions work with a set of values but return only a single value. For example, you can use an aggregate function to count the number of records in a table or to calculate the average price of a product. Table 14-1 lists the most commonly used aggregate functions.

Table 14-1. SQL Aggregate Functions

Function	Description
Avg(fieldname)	Calculates the average of all values in a given numeric field
Sum(fieldname)	Calculates the sum of all values in a given numeric field
Min(fieldname) and Max(fieldname)	Finds the minimum or maximum value in a number field
Count(*)	Returns the number of rows in the result set
Count(DISTINCT fieldname)	Returns the number of unique (and non-null) rows in the result set for the specified field

For example, here's a query that returns a single value—the number of records in the Authors table:

```
SELECT COUNT(*) FROM Authors
```

And here's how you could calculate the total quantity of all sales by adding together the qty field in each record:

```
SELECT SUM(qty) FROM Sales
```

Using the SQL Update Statement

The SQL Update statement selects all the records that match a specified search expression and then modifies them all according to an update expression. At its simplest, the Update statement has the following format:

```
UPDATE [table] SET [update_expression] WHERE [search_condition]
```

Typically, you'll use an Update statement to modify a single record. The following example adjusts the phone column in a single author record. It uses the unique author ID to find the correct row.

```
UPDATE Authors SET phone='408 496-2222' WHERE au_id='172-32-1176'
```

This statement returns the number of affected rows, which in this case is 1. Figure 14-7 shows this example in Visual Studio.

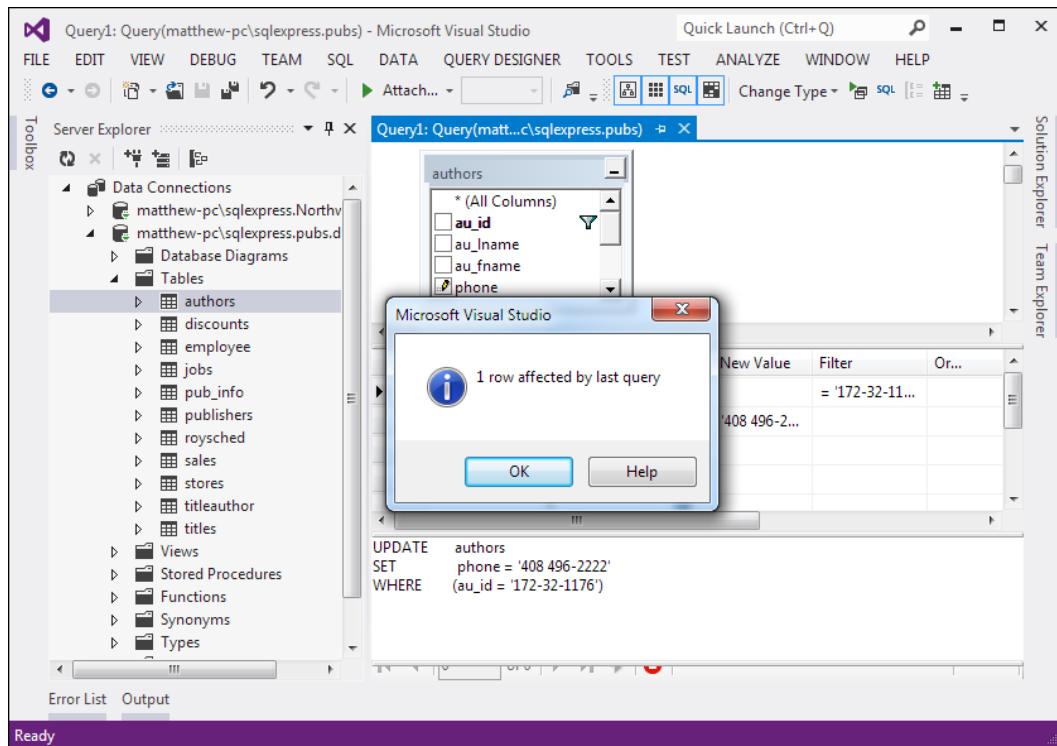


Figure 14-7. Executing an update query in Visual Studio

An Update statement won't display the change that you've made. To do that, you need to request the row by performing another Select statement:

```
SELECT phone FROM Authors WHERE au_id='172-32-1176'
```

As with a Select statement, you can use an Update statement with several criteria:

```
UPDATE Authors SET au_lname='Whiteson', au_fname='John'
WHERE au_lname='White' AND au_fname='Johnson'
```

You can even use the Update statement to update an entire range of matching records. The following example increases the price of every book in the Titles table that was published in 1991 by one dollar:

```
UPDATE Titles SET price=price+1
WHERE pubdate >= '1991/01/01' AND pubdate < '1992/01/01'
```

Using the SQL Insert Statement

The SQL Insert statement adds a new record to a table with the information you specify. It takes the following form:

```
INSERT INTO [table] ([column_list]) VALUES ([value_list])
```

You can provide the information in any order you want, as long as you make sure the list of column names and the list of values correspond exactly:

```
INSERT INTO Authors (au_id, au_lname, au_fname, zip, contract)
VALUES ('998-72-3566', 'Khan', 'John', 84152, 0)
```

This example leaves out some information, such as the city and address, in order to provide a simple example. However, it provides the minimum information that's required to create a new record in the Authors table.

Remember, database tables often have requirements that can prevent you from adding a record unless you fill in all the fields with valid information. Alternatively, some fields may be configured to use a default value if left blank. In the Authors table, some fields are required, and a special format is defined for the ZIP code and author ID.

One feature the Authors table doesn't use is an automatically incrementing identity field. This feature, which is supported in most relational database products, assigns a unique value to a specified field when you perform an insert operation. When you insert a record into a table that has a unique incrementing ID, you shouldn't specify a value for the ID. Instead, allow the database to choose one automatically.

AuTo-InCremenT Fields ARe IndIsPensAble

If you're designing a database, adding an auto-incrementing identity field to every table is considered good design. This is the fastest, easiest, and least error-prone way to assign a unique identification number to every record. Without an automatically generated identity field, you'll need to go to considerable effort to create and maintain your own unique field. Often programmers fall into the trap of using a data field as a unique identifier, such as a social security number (ssN) or a name. This almost always leads to trouble at some inconvenient time far in the future, when you need to add a person who doesn't have an ssN (for example, a foreign national) or you need to account for an ssN or name change (which will cause problems for other related tables, such as a purchase order table that identifies the purchaser by the name or ssN field). A much better approach is to use a unique identifier and have the database engine assign an auto-incrementing unique number to every row automatically.

If you create a table without a unique identification column, you'll have trouble when you need to select a specific row for deletion or updates. Selecting records based on a text field can also lead to problems if the field contains special embedded characters (such as apostrophes). You'll also find it extremely awkward to create table relationships.

Using the SQL Delete Statement

The Delete statement is even easier to use. It specifies criteria for one or more rows that you want to remove. Be careful: after you delete a row, it's gone for good!

```
DELETE FROM [table] WHERE [search_condition]
```

The following example removes a single matching row from the Authors table:

```
DELETE FROM Authors WHERE au_id='172-32-1176'
```

Note If you attempt to run this specific Delete statement, you'll run into a database error. The problem is that this author record is linked to one or more records in the TitleAuthor table. The author record can't be removed unless the linked records are deleted first. (After all, it wouldn't make sense to have a book linked to an author who doesn't exist.)

The Delete and Update commands return a single piece of information: the number of affected records. You can examine this value and use it to determine whether the operation is successful or executed as expected.

The rest of this chapter shows how you can combine SQL with the ADO.NET objects to retrieve and manipulate data in your web applications.

Understanding the Data Provider Model

ADO.NET relies on the functionality in a small set of core classes. You can divide these classes into two groups: those that are used to contain and manage data (such as DataSet, DataTable, DataRow, and DataRelation) and those that are used to connect to a specific data source (such as Connection, Command, and DataReader).

The data container classes are completely generic. No matter what data source you use, after you extract the data, it's stored using the same data container: the specialized DataSet class. Think of the DataSet as playing the same role as a collection or an array—it's a package for data. The difference is that the DataSet is customized for relational data, which means it understands concepts such as rows, columns, and table relationships natively.

The second group of classes exists in several flavors. Each set of data interaction classes is called an ADO.NET *data provider*. Data providers are customized so that each one uses the best-performing way of interacting with its data source. For example, the SQL Server data provider is designed to work with SQL Server. Internally, it uses SQL Server's tabular data stream (TDS) protocol for communicating, thus guaranteeing the best possible performance. If you're using Oracle, you can use an Oracle data provider (which is available at <http://tinyurl.com/2wbsjp6>) instead.

Each provider designates its own prefix for naming classes. Thus, the SQL Server provider includes SqlConnection and SqlCommand classes, and the Oracle provider includes OracleConnection and OracleCommand classes. Internally, these classes work quite differently, because they need to connect to different databases by using different low-level protocols. Externally, however, these classes look quite similar and provide an identical set of basic methods because they implement the same common interfaces. This means your application is shielded from the complexity of different standards and can use the SQL Server provider in the same way the Oracle provider uses it. In fact, you can often translate a block of code for interacting with an SQL Server database into a block of Oracle-specific code just by editing the class names in your code.

In this chapter, you'll use the SQL Server data provider. However, the classes you'll use fall into three key namespaces, as outlined in Table 14-2.

Table 14-2. ADO.NET Namespaces for SQL Server Data Access

Namespace	Purpose
System.Data.SqlClient	Contains the classes you use to connect to a Microsoft SQL Server database and execute commands (such as SqlConnection and SqlCommand).
System.Data.SqlTypes	Contains structures for SQL Server-specific data types such as SqlMoney and SqlDbType. You can use these types to work with SQL Server data types without needing to convert them into the standard .NET equivalents (such as System.Decimal and System.DateTime). These types aren't required, but they do allow you to avoid any potential rounding or conversion problems that could adversely affect data.
System.Data	Contains fundamental classes with the core ADO.NET functionality. These include DataSet and DataRelation, which allow you to manipulate structured relational data. These classes are totally independent of any specific type of database or the way you connect to it.

In the rest of this chapter, you'll consider how to write web page code that uses the classes in these namespaces. First you'll consider the most straightforward approach—direct data access. Then you'll consider disconnected data access, which allows you to retrieve data in the DataSet and cache it for longer periods of time. Both approaches complement each other, and in many web applications you'll use a combination of the two.

Using Direct Data Access

The most straightforward way to interact with a database is to use *direct data access*. When you use direct data access, you're in charge of building an SQL command (like the ones you considered earlier in this chapter) and executing it. You use commands to query, insert, update, and delete information.

When you query data with direct data access, you don't keep a copy of the information in memory. Instead, you work with it for a brief period of time while the database connection is open, and then close the connection as soon as possible. This is different from disconnected data access, where you keep a copy of the data in the DataSet object so you can work with it after the database connection has been closed.

The direct data model is well suited to ASP.NET web pages, which don't need to keep a copy of their data in memory for long periods of time. Remember, an ASP.NET web page is loaded when the page is requested and shut down as soon as the response is returned to the user. That means a page typically has a lifetime of only a few seconds (if that).

Note Although AsP.NET web pages don't need to store data in memory for ordinary data management, they just might use this technique to optimize performance. For example, you could get the product catalog database once, and keep that data in memory on the web server so you can reuse it when someone else visits the same page. This technique is called *caching*, and you'll learn to use it in Chapter 23.

To query information with simple data access, follow these steps:

1. Create Connection, Command, and DataReader objects.
2. Use the DataReader to retrieve information from the database, and display it in a control on a web form.
3. Close your connection.
4. Send the page to the user. At this point, the information your user sees and the information in the database no longer have any connection, and all the ADO.NET objects have been destroyed.

To add or update information, follow these steps:

1. Create new Connection and Command objects.
2. Execute the Command (with the appropriate SQL statement).

This chapter demonstrates both of these approaches. Figure 14-8 shows a high-level look at how the ADO.NET objects interact to make direct data access work.

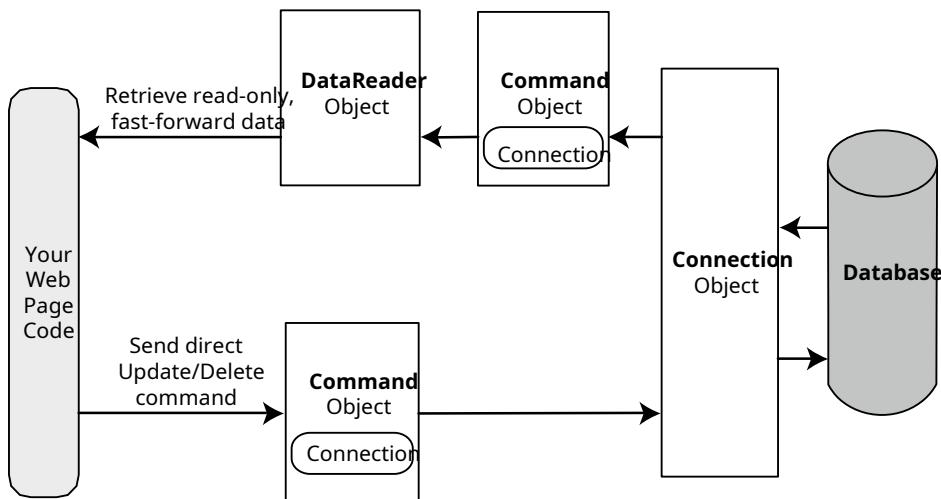


Figure 14-8. Direct data access with ADO.NET

Before continuing, make sure you import the ADO.NET namespaces. In this chapter, we assume you're using the SQL Server provider, in which case you need these two namespace imports:

```
using System.Data;
using System.Data.SqlClient;
```

Creating a Connection

Before you can retrieve or update data, you need to make a connection to the data source. Generally, connections are limited to some fixed number, and if you exceed that number (either because you run out of licenses or because your database server can't accommodate the user load), attempts to create new connections will fail. For that reason, you should try to hold a connection open for as short a time as possible. You should also write your database code inside a try/catch error-handling structure so you can respond if an error does occur, and make sure you close the connection even if you can't perform all your work.

When creating a `Connection` object, you need to specify a value for its `ConnectionString` property. This `ConnectionString` defines all the information the computer needs to find the data source, log in, and choose an initial database. Out of all the details in the examples in this chapter, the `ConnectionString` is the one value you might have to tweak before it works for the database you want to use. Luckily, it's quite straightforward. Here's an example that uses a connection string to connect to SQL Server:

```
SqlConnection myConnection = new SqlConnection();
myConnection.ConnectionString = "Data Source=localhost;" +
"Initial Catalog=Pubs;Integrated Security=SSPI";
```

If you're using SQL Server Express, your connection string will use the instance name, as shown here:

```
SqlConnection myConnection = new SqlConnection();
myConnection.ConnectionString = @"Data Source=(localdb)\v11.0;" +
"Initial Catalog=Pubs;Integrated Security=SSPI";
```

Note Ordinarily, the C# compiler assumes that every backslash designates the start of a special character sequence. This can cause issues when you use connection strings in code. There are two ways to deal with this problem. You can use two backslashes instead of one, as in `(localdb)\v11.0`. Or you can put the `@` character before the entire string, as in `@"(localdb)\v11.0"`. However, if you define the connection string configuration file, as described in the next section, you don't need to take either of these steps. That's because you're no longer dealing with pure C# code, so a single backslash is fine.

Exploring the Connection String

The connection string is actually a series of distinct pieces of information separated by semicolons (`:`). Each separate piece of information is known as a *connection string property*.

The following list describes some of the most commonly used connection string properties, including the three properties used in the preceding example:

Data source: This indicates the name of the server where the data source is located. If the server is on the same computer that hosts the ASP.NET site, `localhost` is sufficient. But if you're using SQL Server Express, the connection string changes. For SQL Server Express LocalDB (the test version included with Visual Studio), the data source is `(localdb)\v11.0`. For the full version of SQL Server Express, which you might use in a deployed application, it's `localhost\SQLExpress` (or `ServerName\SQLExpress` if the database is running on another computer). You'll also see this written with a period, as `.\SQLExpress`, which is equivalent.

Initial catalog: This is the name of the database that this connection will be accessing. It's only the "initial" database, because you can change it later by using the `Connection.ChangeDatabase()` method.

Integrated security: This indicates that you want to connect to SQL Server by using the Windows user account that's running the web page code, provided you supply a value of SSPI (which stands for *Security Support Provider Interface*). Alternatively, you can supply a user ID and password that's defined in the database for SQL Server authentication, although this method is less secure and generally discouraged.

ConnectionTimeout: This determines the number of seconds your code will wait before generating an error if it cannot establish a database connection. Our example connection string doesn't set the `ConnectionTimeout`, so the default of 15 seconds is used. You can use 0 to specify no limit, but this is a bad idea. This means that, theoretically, the code could be held up indefinitely while it attempts to contact the server.

You can set some other, lesser-used options for a connection string. For more information, refer to the Visual Studio Help. Look under the appropriate `Connection` class (such as `SqlConnection` or `OleDbConnection`), because there are subtle differences in connection string properties for each type of `Connection` class.

Using Windows Authentication

The previous example uses *integrated Windows authentication*, which is the default security standard for new SQL Server installations. You can also use *SQL Server authentication*. In this case, you will explicitly place the user ID and password information in the connection string. However, SQL Server authentication is disabled by default in modern versions of SQL Server, because it's not considered to be as secure.

Here's the lowdown on both types of authentication:

- With SQL Server authentication, SQL Server maintains its own user account information in the database. It uses this information to determine whether you are allowed to access specific parts of a database.
- With integrated Windows authentication, SQL Server automatically uses the Windows account information for the currently logged-in process. In the database, it stores information about what database privileges each user should have.

Tip You can set the type of authentication that your sQL server uses by using a tool such as sQL server management studio. Just right-click your server in the tree and select Properties. Choose the security tab to change the type of authentication. You can choose either Windows Only (for the tightest security) or sQL and Windows, which allows both Windows authentication and sQL server authentication. This option is also known as *mixed-mode authentication*.

For Windows authentication to work, the currently logged-on Windows user must have the required authorization to access the SQL database. This isn't a problem while you test your websites, because Visual Studio launches your web applications by using your user account. However, when you deploy your application to a web server running IIS, you might run into trouble. In this situation, all ASP.NET code is run by a more limited user account that might not have the rights to access the database. Although the exact user depends on your version of IIS (see the discussion in Chapter 26), the best approach is usually to grant access to the IIS_IUSRS group.

Making User Instance Connections

Every database server stores a master list of all the databases that you've installed on it. This list includes the name of each database and the location of the files that hold the data. When you create a database (for example, by running a script or using a management tool), the information about that database is added to the master list. When you connect to the database, you specify the database name by using the Initial Catalog value in the connection string.

Note If you haven't made any changes to your database configuration, sQL server will quietly tuck the newly created databases into a directory such as c:\Program Files\microsoft sQL server\mssQL10.sQLEXP\mssQL\Data (although the exact path depends on the version of sQL server you're using). Each database has at least two files—an .mdf file with the actual data and an .ldf file that stores the database log. Of course, database professionals have a variety of techniques and tricks for managing database storage, and can easily store databases in different locations, create multiple data files, and so on. The important detail to realize is that ordinary database files are stored by your database server, and they aren't a part of your web application directory.

Interestingly, SQL Server Express has a feature that lets you bypass the master list and connect directly to any database file, even if it's not in the master list of databases. This feature is called *user instances*. Oddly enough, this feature isn't available in the full edition of SQL Server.

To use this feature, you need to set the User Instances value to True (in the connection string) and supply the file name of the database you want to connect to with the AttachDBFilename value. You don't supply an Initial Catalog value.

Here's an example connection string that uses this approach:

```
myConnection.ConnectionString = @"Data Source=(localdb)\v11.0;" +
 @"User Instance=True;AttachDBFilename=|DataDirectory|\Northwind.mdf;" +
 "Integrated Security=True";
```

There's another trick here. The file name starts with |DataDirectory|. This automatically points to the App_Data folder inside your web application directory. This way, you don't need to supply a full file path, which might not remain valid when you move the web application to a web server. Instead, ADO.NET will always look in the App_Data directory for a file named Northwind.mdf.

User instances are a handy feature if you have a web server that hosts many different web applications that use databases and these databases are frequently being added and removed. However, because the database isn't in the master list, you won't see it in any administrative tools (although most administrative tools will still let you connect to it manually, by pointing out the right file location). But remember, this quirky but interesting feature is available in SQL Server Express only—you won't find it in the full version of SQL Server.

VIsuAl sTudIo's suPPoRT FoR useR InSTAnCe dATAbAsEs

Visual studio provides two handy features that make it easier to work with databases in the App_Data folder.

First, Visual studio gives you a nearly effortless way to create new databases. simply choose Website New Item. Then pick SQL server Database from the list of templates, choose a file name for your database and click OK. The .mdf and .ldf files for the new database will be placed in the App_Data folder, and you can see them in the solution Explorer. Initially, they'll be blank, so you'll need to add the tables you want. The easiest way to do this is to right-click the Tables group in the server Explorer and choose Add Table.

Visual studio also simplifies your life with its automatic server Explorer support. When you open a web application, Visual studio automatically adds a data connection to the server Explorer window for each database that it finds in the App_Data folder. To jump to a specific data connection in a hurry, just double-click the .mdf file for the database in the solution Explorer.

using the server Explorer, you can create tables, edit data, and execute commands, all without leaving the comfort of Visual studio. (For more information about executing commands with the server Explorer, see the "understanding SQL Basics" section earlier in this chapter.)

Storing the Connection String

Typically, all the database code in your application will use the same connection string. Therefore, it usually makes the most sense to store a connection string in a class member variable or, even better, a configuration file.

You can also create a Connection object and supply the connection string in one step by using a dedicated constructor:

```
SqlConnection myConnection = new SqlConnection(connectionString);
// myConnection.ConnectionString is now set to connectionString.
```

You don't need to hard-code a connection string. The `<connectionStrings>` section of the `web.config` file is a handy place to store your connection strings. Here's an example:

```
<configuration>
<connectionStrings>
  <add name="Pubs" connectionString=
    "Data Source=localhost;Initial Catalog=Pubs;Integrated Security=SSPI"/>
</connectionStrings>
...
</configuration>
```

You can then retrieve your connection string by name. First import the `System.Web.Configuration` namespace. Then you can use code like this:

```
string connectionString =
  WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
```

This approach helps ensure that all your web pages are using the same connection string. It also makes it easy for you to change the connection string for an application, without needing to edit the code in multiple pages. The examples in this chapter all store their connection strings in the `web.config` file in this way.

Making the Connection

After you've created your connection (as described in the previous section), you're ready to use it.

Before you can perform any database operations, you need to explicitly open your connection:

```
myConnection.Open();
```

To verify that you have successfully connected to the database, you can try displaying some basic connection information. The following example writes some basic information to a Label control named `lblInfo` (see Figure 14-9).

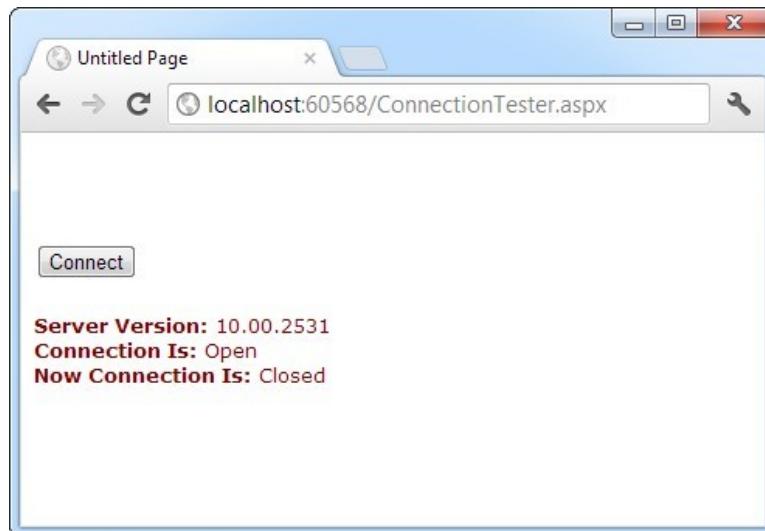


Figure 14-9. Testing your connection

Here's the code with basic error handling:

```
// Define the ADO.NET Connection object.
string connectionString =
    WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
SqlConnection myConnection = new SqlConnection(connectionString);

try
{
    // Try to open the connection.
    myConnection.Open();
    lblInfo.Text = "<b>Server Version:</b> " + myConnection.ServerVersion;
    lblInfo.Text += "<br /><b>Connection Is:</b> " +
        myConnection.State.ToString();
}
catch (Exception err)
{
    // Handle an error by displaying the information.
    lblInfo.Text = "Error reading the database. ";
    lblInfo.Text += err.Message;
}
finally
{
    // Either way, make sure the connection is properly closed.
    // (Even if the connection wasn't opened successfully,
    // calling Close() won't cause an error.)
    myConnection.Close();
    lblInfo.Text += "<br /><b>Now Connection Is:</b> ";
    lblInfo.Text += myConnection.State.ToString();
}
```

After you use the Open() method, you have a live connection to your database. One of the most fundamental principles of data access code is that you should reduce the amount of time you hold a connection open as much as possible. Imagine that as soon as you open the connection, you have a live, ticking time bomb. You need to get in, retrieve your data, and throw the connection away as quickly as possible in order to ensure that your site runs efficiently.

Closing a connection is just as easy, as shown here:

```
myConnection.Close();
```

Another approach is to use the using statement. The using statement declares that you are using a disposable object for a short period of time. As soon as you finish using that object and the using block ends, the common language runtime will release it immediately by calling the Dispose() method. Here's the basic structure of the using block:

```
using (object)
{
    ...
}
```

It just so happens that calling the Dispose() method of a connection object is equivalent to calling Close() and then discarding the connection object from memory. That means you can shorten your database code with the help of a using block. The best part is that you don't need to write a finally block—the using statement releases the object you're using even if you exit the block as the result of an unhandled exception.

Here's how you could rewrite the earlier example with a using block:

```
// Define the ADO.NET Connection object.
string connectionString =
    WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
SqlConnection myConnection = new SqlConnection(connectionString);

try
{
    using (myConnection)
    {
        // Try to open the connection.
        myConnection.Open();
        lblInfo.Text = "<b>Server Version:</b> " + myConnection.ServerVersion;
        lblInfo.Text += "<br /><b>Connection Is:</b> " +
            myConnection.State.ToString();
    }
}
catch (Exception err)
{
    // Handle an error by displaying the information.
    lblInfo.Text = "Error reading the database. ";
    lblInfo.Text += err.Message;
}

lblInfo.Text += "<br /><b>Now Connection Is:</b> ";
lblInfo.Text += myConnection.State.ToString();
```

There's one difference in the way this code is implemented as compared to the previous example. The error-handling code wraps the using block. As a result, if an error occurs, the database connection is closed first, and *then* the exception-handling code is triggered. In the first example, the error-handling code responded first, and then the finally block closed the connection afterward. Obviously, this rewrite is a bit better, as it's always good to close database connections as soon as possible.

Using the Select Command

The Connection object provides a few basic properties that supply information about the connection, but that's about all. To actually retrieve data, you need a few more ingredients:

- An SQL statement that selects the information you want
- A Command object that executes the SQL statement
- A DataReader or DataSet object to access the retrieved records

Command objects represent SQL statements. To use a Command object, you define it, specify the SQL statement you want to use, specify an available connection, and execute the command. To ensure good database performance, you should open your connection just before you execute your command and close it as soon as the command is finished.

You can use one of the earlier SQL statements, as shown here:

```
SqlCommand myCommand = new SqlCommand();
myCommand.Connection = myConnection;
myCommand.CommandText = "SELECT * FROM Authors ORDER BY au_Iname ";
```

Or you can use the constructor as a shortcut:

```
SqlCommand myCommand = new SqlCommand(
    "SELECT * FROM Authors ORDER BY au_lname ", myConnection);
```

Note It's also a good idea to dispose of the Command object when you're finished, although it isn't as closing the Connection object.

Using the DataReader

After you've defined your command, you need to decide how you want to use it. The simplest approach is to use a DataReader, which allows you to quickly retrieve all your results. The DataReader uses a live connection and should be used quickly and then closed. The DataReader is also extremely simple. It supports fast-forward-only read-only access to your results, which is generally all you need when retrieving information. Because of the DataReader's optimized nature, it provides better performance than the DataSet. It should always be your first choice for direct data access.

Before you can use a DataReader, make sure you've opened the connection:

```
myConnection.Open();
```

To create a DataReader, you use the ExecuteReader() method of the command object, as shown here:

```
// You don't need the new keyword, as the Command will create the DataReader.
SqlDataReader myReader;
myReader = myCommand.ExecuteReader();
```

These two lines of code define a variable for a DataReader and then create it by executing the command. After you have the reader, you retrieve a single row at a time by using the Read() method:

```
myReader.Read(); // The first row in the result set is now available.
```

You can then access the values in the current row by using the corresponding field names. The following example adds an item to a list box with the first name and last name for the current row:

```
lstNames.Items.Add(myReader["au_lname"] + ", " + myReader["au_fname"]);
```

To move to the next row, use the Read() method again. If this method returns True, a row of information has been successfully retrieved. If it returns False, you've attempted to read past the end of your result set. There is no way to move backward to a previous row.

As soon as you've finished reading all the results you need, close the DataReader and Connection:

```
myReader.Close();
myConnection.Close();
```

Putting It All Together

The next example demonstrates how you can use all the ADO.NET ingredients together to create a simple application that retrieves information from the Authors table. You can select an author record by last name by using a drop-down list box, as shown in Figure 14-10.

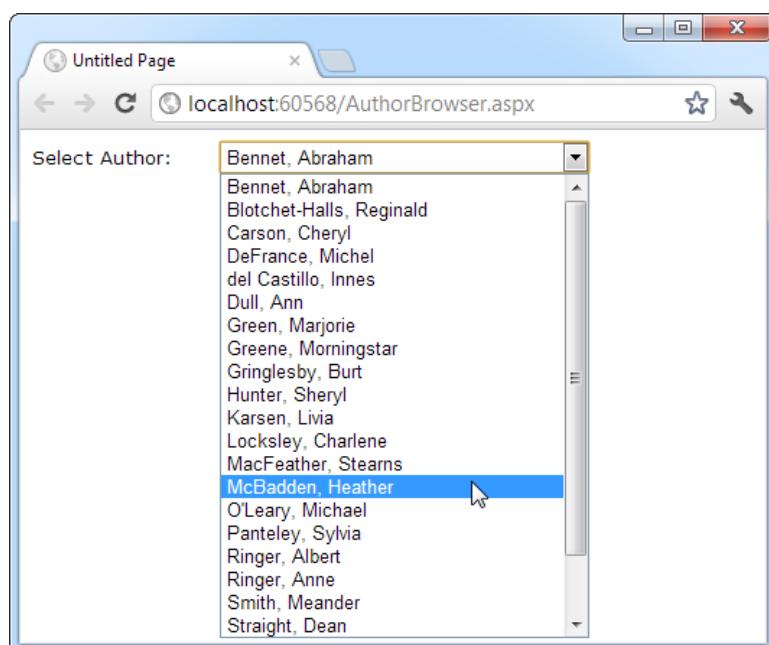


Figure 14-10. Selecting an author

The full record is then retrieved and displayed in a simple label, as shown in Figure 14-11.

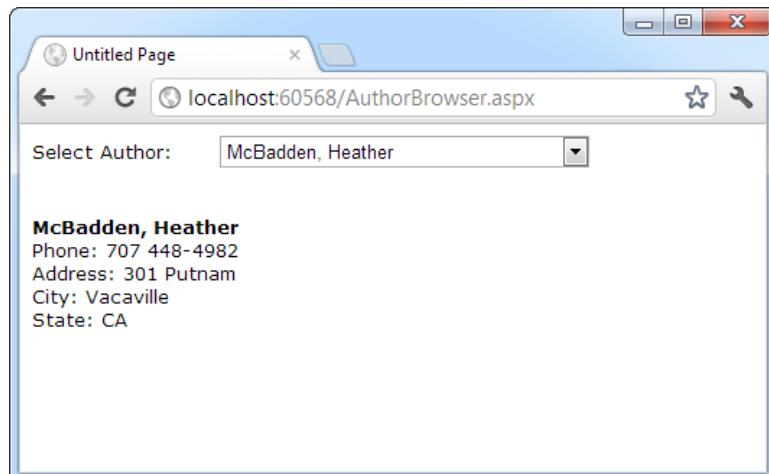


Figure 14-11. Author information

Filling the List Box

To start, the connection string is defined as a private variable for the page class and retrieved from the connection string:

```
private string connectionString =
    WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
```

The list box is filled when the Page.Load event occurs. Because the list box is set to persist its view state information, this information needs to be retrieved only once—the first time the page is displayed. It will be ignored on all postbacks.

Here's the code that fills the list from the database:

```
protected void Page_Load(Object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        FillAuthorList();
    }
}

private void FillAuthorList()
{
    lstAuthor.Items.Clear();

    // Define the Select statement.
    // Three pieces of information are needed: the unique id
    // and the first and last name.
    string selectSQL = "SELECT au_lname, au_fname, au_id FROM Authors";

    // Define the ADO.NET objects.
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataReader reader;

    // Try to open database and read information.
    try
    {
        con.Open();
        reader = cmd.ExecuteReader();

        // For each item, add the author name to the displayed
        // list box text, and store the unique ID in the Value property.
        while (reader.Read())
        {
            ListItem newItem = new ListItem();
            newItem.Text = reader["au_lname"] + ", " + reader["au_fname"];
            newItem.Value = reader["au_id"].ToString();
            lstAuthor.Items.Add(newItem);
        }
        reader.Close();
    }
    catch (Exception err)
```

```

{
    lblResults.Text = "Error reading list of names. ";
    lblResults.Text += err.Message;
}
finally
{
    con.Close();
}
}

```

This example looks more sophisticated than the previous bite-sized snippets in this chapter, but it really doesn't introduce anything new. It uses the standard Connection, Command, and DataReader objects. The Connection is opened inside an error-handling block so your page can handle any unexpected errors and provide information. A finally block makes sure the connection is properly closed, even if an error occurs.

The actual code for reading the data uses a loop. With each pass, the Read() method is called to get another row of information. When the reader has read all the available information, this method will return false, the loop condition will evaluate to false, and the loop will end gracefully.

The unique ID (the value in the au_id field) is stored in the Value property of the list box for reference later. This is a crucial ingredient that is needed to allow the corresponding record to be queried again. If you tried to build a query using the author's name, you would need to worry about authors with the same name. You would also have the additional headache of invalid characters (such as the apostrophe in O'Leary) that would invalidate your SQL statement.

Retrieving the Record

The record is retrieved as soon as the user changes the selection in the list box. To make this possible, the AutoPostBack property of the list box is set to true so that its change events are detected automatically.

```

protected void lstAuthor_SelectedIndexChanged(Object sender, EventArgs e)
{
    // Create a Select statement that searches for a record
    // matching the specific author ID from the Value property.
    string selectSQL;
    selectSQL = "SELECT * FROM Authors ";
    selectSQL += "WHERE au_id=" + lstAuthor.SelectedItem.Value + "";

    // Define the ADO.NET objects.
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataReader reader;

    // Try to open database and read information.
    try
    {
        con.Open();
        reader = cmd.ExecuteReader();
        reader.Read();

        // Build a string with the record information,
        // and display that in a label.
        StringBuilder sb = new StringBuilder();
        sb.Append("<b>");

```

```

sb.Append(reader["au_lname"]);
sb.Append(", ");
sb.Append(reader["au_fname"]);
sb.Append("</b><br />");
sb.Append("Phone: ");
sb.Append(reader["phone"]);
sb.Append("<br />");
sb.Append("Address: ");
sb.Append(reader["address"]);
sb.Append("<br />");
sb.Append("City: ");
sb.Append(reader["city"]);
sb.Append("<br />");
sb.Append("State: ");
sb.Append(reader["state"]);
sb.Append("<br />");
lblResults.Text = sb.ToString();

reader.Close();
}
catch (Exception err)
{
    lblResults.Text = "Error getting author. ";
    lblResults.Text += err.Message;
}
finally
{
    con.Close();
}
}

```

The process is similar to the procedure used to retrieve the last names. There are only a couple of differences:

- The code dynamically creates an SQL statement based on the selected item in the drop-down list box. It uses the Value property of the selected item, which stores the unique identifier. This is a common (and useful) technique.
- Only one record is read. The code assumes that only one author has the matching au_id, which is reasonable because this field is unique.

Note This example shows how ADO.NET works to retrieve a simple result set. Of course, ADO.NET also includes handy controls that go beyond this generic level and let you provide full-featured grids with sorting and filtering. These controls are described in Chapter 15 and Chapter 16. For now, you should concentrate on understanding the fundamentals of ADO.NET and how it works with data.

Updating Data

Now that you understand how to retrieve data, it isn't much more complicated to perform simple insert, update, and delete operations. Once again, you use the Command object, but this time you don't need a DataReader

because no results will be retrieved. You also don't use an SQL Select command. Instead, you use one of three new SQL commands: Update, Insert, or Delete.

To execute an Update, Insert, or Delete statement, you need to create a Command object. You can then execute the command with the ExecuteNonQuery() method. This method returns the number of rows that were affected, which allows you to check your assumptions. For example, if you attempt to update or delete a record and are informed that no records were affected, you probably have an error in your Where clause that is preventing any records from being selected. (If, on the other hand, your SQL command has a syntax error or attempts to retrieve information from a nonexistent table, an exception will occur.)

Displaying Values in Text Boxes

Before you can update and insert records, you need to make a change to the previous example. Instead of displaying the field values in a single, fixed label, you need to show each detail in a separate text box. Figure 14-12 shows the revamped page. It includes two new buttons that allow you to update the record (Update) or delete it (Delete), and two more that allow you to begin creating a new record (Create New) and then insert it (Insert New).

The screenshot shows a web browser window titled "Untitled Page" with the URL "localhost:60568/AuthorManager.aspx". The page contains a form for managing author records. At the top, there is a dropdown menu labeled "Select Author:" containing the value "Blotchet-Halls, Reginald". Next to it are "Update" and "Delete" buttons. Below this, there is a link "Or:" followed by "Create New" and "Insert New" buttons. The main area of the form contains the following fields:

Unique ID:	648-92-1872	(required: ####-##-##### form)
First Name:	Reginald	
Last Name:	Blotchet-Halls	
Phone:	503 745-6402	
Address:	55 Hillsdale Bl.	
City:	Corvallis	
State:	OR	
Zip Code:	97330	(required: any five digits)
Contract:	<input checked="" type="checkbox"/>	

Figure 14-12. A more advanced author manager

The record selection code is identical from an ADO.NET perspective, but it now uses the individual text boxes:

```

protected void lstAuthor_SelectedIndexChanged(Object sender, EventArgs e)
{
    // Define ADO.NET objects.
    string selectSQL;
    selectSQL = "SELECT * FROM Authors ";
    selectSQL += "WHERE au_id=" + lstAuthor.SelectedItem.Value + "";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataReader reader;

    // Try to open database and read information.
    try
    {
        con.Open();
        reader = cmd.ExecuteReader();
        reader.Read();

        // Fill the controls.
        txtID.Text = reader["au_id"].ToString();
        txtFirstName.Text = reader["au_fname"].ToString();
        txtLastName.Text = reader["au_lname"].ToString();
        txtPhone.Text = reader["phone"].ToString();
        txtAddress.Text = reader["address"].ToString();
        txtCity.Text = reader["city"].ToString();
        txtState.Text = reader["state"].ToString();
        txtZip.Text = reader["zip"].ToString();
        chkContract.Checked = (bool)reader["contract"];
        reader.Close();
        lblStatus.Text = "";
    }
    catch (Exception err)
    {
        lblStatus.Text = "Error getting author. ";
        lblStatus.Text += err.Message;
    }
    finally
    {
        con.Close();
    }
}

```

To see the full code, refer to the online samples for this chapter. If you play with the example at length, you'll notice that it lacks a few niceties that would be needed in a professional website. For example, when creating a new record, the name of the last selected user is still visible, and the Update and Delete buttons are still active, which can lead to confusion or errors. A more sophisticated user interface could prevent these problems by disabling inapplicable controls (perhaps by grouping them in a Panel control) or by using separate pages. In this case, however, the page is useful as a quick way to test some basic data access code.

Adding a Record

To start adding a new record, click Create New to clear all the text boxes. Technically, this step isn't required, but it simplifies the user's life:

```
protected void cmdNew_Click(Object sender, EventArgs e)
{
    txtID.Text = "";
    txtFirstName.Text = "";
    txtLastName.Text = "";
    txtPhone.Text = "";
    txtAddress.Text = "";
    txtCity.Text = "";
    txtState.Text = "";
    txtZip.Text = "";
    chkContract.Checked = false;

    lblStatus.Text = "Click Insert New to add the completed record.";
}
```

The Insert New button triggers the ADO.NET code that uses a dynamically generated Insert statement to insert the finished record:

```
protected void cmdInsert_Click(Object sender, EventArgs e)
{
    // Perform user-defined checks.
    // Alternatively, you could use RequiredFieldValidator controls.
    if (txtID.Text == "" || txtFirstName.Text == "" || txtLastName.Text == "")
    {
        lblStatus.Text = "Records require an ID, first name, and last name.";
        return;
    }

    // Define ADO.NET objects.
    string insertSQL;
    insertSQL = "INSERT INTO Authors (";
    insertSQL += "au_id, au_fname, au_lname, ";
    insertSQL += "phone, address, city, state, zip, contract) ";
    insertSQL += "VALUES (";
    insertSQL += txtID.Text + ", ";
    insertSQL += txtFirstName.Text + ", ";
    insertSQL += txtLastName.Text + ", ";
    insertSQL += txtPhone.Text + ", ";
    insertSQL += txtAddress.Text + ", ";
    insertSQL += txtCity.Text + ", ";
    insertSQL += txtState.Text + ", ";
    insertSQL += txtZip.Text + ", ";
    insertSQL += Convert.ToInt16(chkContract.Checked) + ")";
}
```

```
SqlConnection con = new SqlConnection(connectionString);
SqlCommand cmd = new SqlCommand(insertSQL, con);

// Try to open the database and execute the update.
int added = 0;
try
{
    con.Open();
    added = cmd.ExecuteNonQuery();
    lblStatus.Text = added.ToString() + " records inserted.";
}
catch (Exception err)
{
    lblStatus.Text = "Error inserting record. ";
    lblStatus.Text += err.Message;
}
finally
{
    con.Close();
}

// If the insert succeeded, refresh the author list.
if (added > 0)
{
    FillAuthorList();
}
```

If the insert fails, the problem will be reported to the user in a rather unfriendly way (see Figure 14-13). This is typically the result of not specifying valid values. If the insert operation is successful, the page is updated with the new author list.

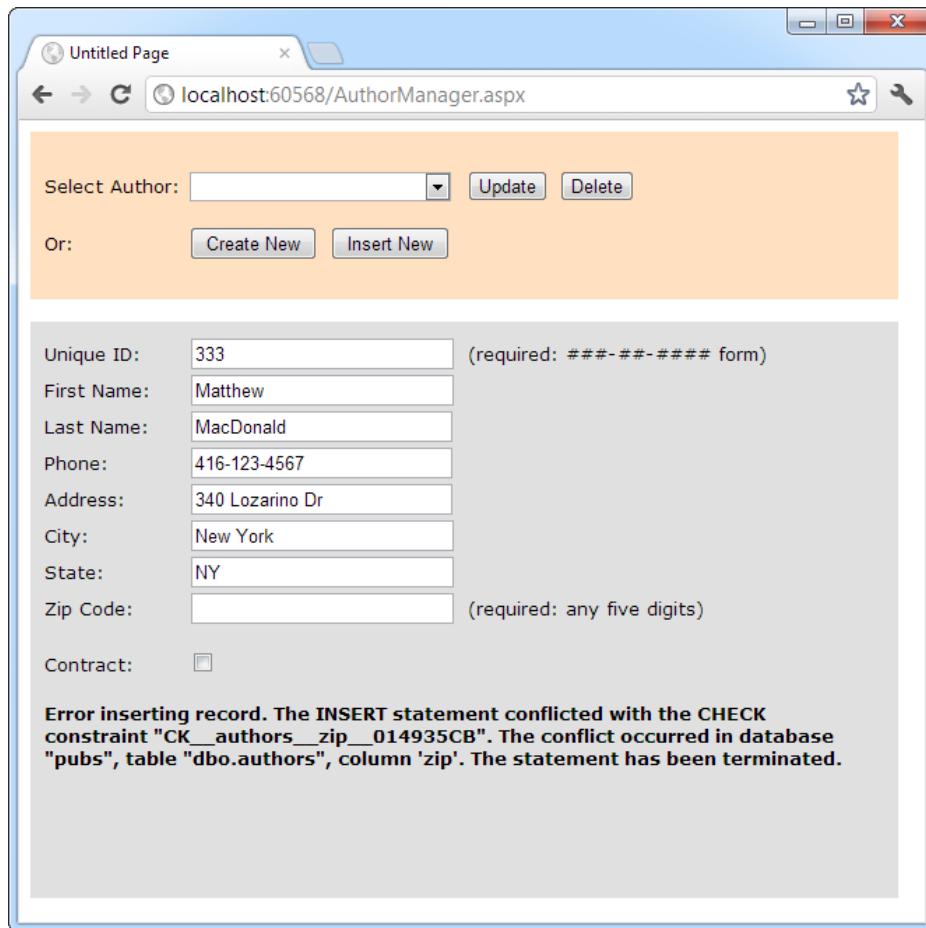


Figure 14-13. A failed insertion

Note In a more polished application, you would use validators (as shown in Chapter 9) and provide meaningful error messages. You should never display the detailed database error information shown in Figure 14-13 because it could give valuable information to malicious users.

Creating More Robust Commands

The previous example performed its database work by using a dynamically pasted-together SQL string. This off-the-cuff approach is great for quickly coding database logic and is easy to understand. However, it has two potentially serious drawbacks:

- Users may accidentally enter characters that will affect your SQL statement. For example, if a value contains an apostrophe ('), the pasted-together SQL string will no longer be valid.

- Users might *deliberately* enter characters that will affect your SQL statement. Examples include using the single apostrophe to close a value prematurely and then following the value with additional SQL code.

The second of these is known as an *SQL injection attack*, which facilitates an amazingly wide range of exploits. Crafty users can use SQL injection attacks to do anything, from returning additional results (such as the orders placed by other customers), to even executing additional SQL statements (such as deleting every record in another table in the same database). In fact, SQL Server includes a special system stored procedure that allows users to execute arbitrary programs on the computer, so this vulnerability can be extremely serious.

You could address these problems by carefully validating the supplied input and checking for dangerous characters such as apostrophes. One approach is to sanitize your input by doubling all apostrophes in the user input (in other words, replace ' with '). Here's an example:

```
string authorID = txtID.Text.Replace("''", "'''");
```

A much more robust and convenient approach is to use a *parameterized command*. A parameterized command is one that replaces hard-coded values with placeholders. The placeholders are then added separately and automatically encoded.

For example, this SQL statement:

```
SELECT * FROM Customers WHERE CustomerID = 'ALFKI'
```

would become this:

```
SELECT * FROM Customers WHERE CustomerID = @CustomerID
```

You then need to add a Parameter object for each parameter in the Command.Parameters collection.

The following example rewrites the insert code of the author manager example with a parameterized command:

```
protected void cmdInsert_Click(Object sender, EventArgs e)
{
    // Perform user-defined checks.
    if (txtID.Text == "" || txtFirstName.Text == "" || txtLastName.Text == "")
    {
        lblStatus.Text = "Records require an ID, first name, and last name.";
        return;
    }

    // Define ADO.NET objects.
    string insertSQL;
    insertSQL = "INSERT INTO Authors (";
    insertSQL += "au_id, au_fname, au_lname, ";
    insertSQL += "phone, address, city, state, zip, contract)";
    insertSQL += "VALUES (";
    insertSQL += "@au_id, @au_fname, @au_lname, ";
    insertSQL += "@phone, @address, @city, @state, @zip, @contract)";

    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(insertSQL, con);

    // Add the parameters.
    cmd.Parameters.AddWithValue("@au_id", txtID.Text);
    cmd.Parameters.AddWithValue("@au_fname", txtFirstName.Text);
    cmd.Parameters.AddWithValue("@au_lname", txtLastName.Text);
```

```

cmd.Parameters.AddWithValue("@phone", txtPhone.Text);
cmd.Parameters.AddWithValue("@address", txtAddress.Text);
cmd.Parameters.AddWithValue("@city", txtCity.Text);
cmd.Parameters.AddWithValue("@state", txtState.Text);
cmd.Parameters.AddWithValue("@zip", txtZip.Text);
cmd.Parameters.AddWithValue("@contract", chkContract.Checked);

// Try to open the database and execute the update.
int added = 0;
try
{
    con.Open();
    added = cmd.ExecuteNonQuery();
    lblStatus.Text = added.ToString() + " record inserted.";
}
catch (Exception err)
{
    lblStatus.Text = "Error inserting record. ";
    lblStatus.Text += err.Message;
}
finally
{
    con.Close();
}

// If the insert succeeded, refresh the author list.
if (added > 0)
{
    FillAuthorList();
}
}

```

Now that the values have been moved out of the SQL command and to the Parameters collection, there's no way that a misplaced apostrophe or scrap of SQL can cause a problem.

Caution For basic security, *always* use parameterized commands. many of the most infamous attacks e-commerce websites weren't fueled by hard-core hacker knowledge but used simple sQL injection to manipulate values in web pages or query strings.

Updating a Record

When the user clicks the Update button, the information in the text boxes is applied to the database as follows:

```

protected void cmdUpdate_Click(Object sender, EventArgs e)
{
    // Define ADO.NET objects.
    string updateSQL;
    updateSQL = "UPDATE Authors SET ";

```

```
updateSQL += "au_fname=@au_fname, au_lname=@au_lname, ";
updateSQL += "phone=@phone, address=@address, city=@city, state=@state, ";
updateSQL += "zip=@zip, contract=@contract ";
updateSQL += "WHERE au_id=@au_id_original";

SqlConnection con = new SqlConnection(connectionString);
SqlCommand cmd = new SqlCommand(updateSQL, con);

// Add the parameters.
cmd.Parameters.AddWithValue("@au_fname", txtFirstName.Text);
cmd.Parameters.AddWithValue("@au_lname", txtLastName.Text);
cmd.Parameters.AddWithValue("@phone", txtPhone.Text);
cmd.Parameters.AddWithValue("@address", txtAddress.Text);
cmd.Parameters.AddWithValue("@city", txtCity.Text);
cmd.Parameters.AddWithValue("@state", txtState.Text);
cmd.Parameters.AddWithValue("@zip", txtZip.Text);
cmd.Parameters.AddWithValue("@contract", chkContract.Checked);
cmd.Parameters.AddWithValue("@au_id_original",
    lstAuthor.SelectedItem.Value);

// Try to open database and execute the update.
int updated = 0;
try
{
    con.Open();
    updated = cmd.ExecuteNonQuery();
    lblStatus.Text = updated.ToString() + " record updated.";
}
catch (Exception err)
{
    lblStatus.Text = "Error updating author. ";
    lblStatus.Text += err.Message;
}
finally
{
    con.Close();
}

// If the update succeeded, refresh the author list.
if (updated > 0)
{
    FillAuthorList();
}
```

Deleting a Record

When the user clicks the Delete button, the author information is removed from the database. The number of affected records is examined, and if the delete operation was successful, the FillAuthorList() function is called to refresh the page.

```
protected void cmdDelete_Click(Object sender, EventArgs e)
{
    // Define ADO.NET objects.
    string deleteSQL;
    deleteSQL = "DELETE FROM Authors ";
    deleteSQL += "WHERE au_id=@au_id";

    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(deleteSQL, con);
    cmd.Parameters.AddWithValue("@au_id ", lstAuthor.SelectedItem.Value);

    // Try to open the database and delete the record.
    int deleted = 0;
    try
    {
        con.Open();
        deleted = cmd.ExecuteNonQuery();
    }
    catch (Exception err)
    {
        lblStatus.Text = "Error deleting author. ";
        lblStatus.Text += err.Message;
    }
    finally
    {
        con.Close();
    }

    // If the delete succeeded, refresh the author list.
    if (deleted > 0)
    {
        FillAuthorList();
    }
}
```

Interestingly, delete operations rarely succeed with the records in the pubs database, because they have corresponding child records linked in another table of the pubs database. Specifically, each author can have one or more related book titles. Unless the author's records are removed from the TitleAuthor table first, the author cannot be deleted. Because of the careful error handling used in the previous example, this problem is faithfully reported in your application (see Figure 14-14) and doesn't cause any real problems.

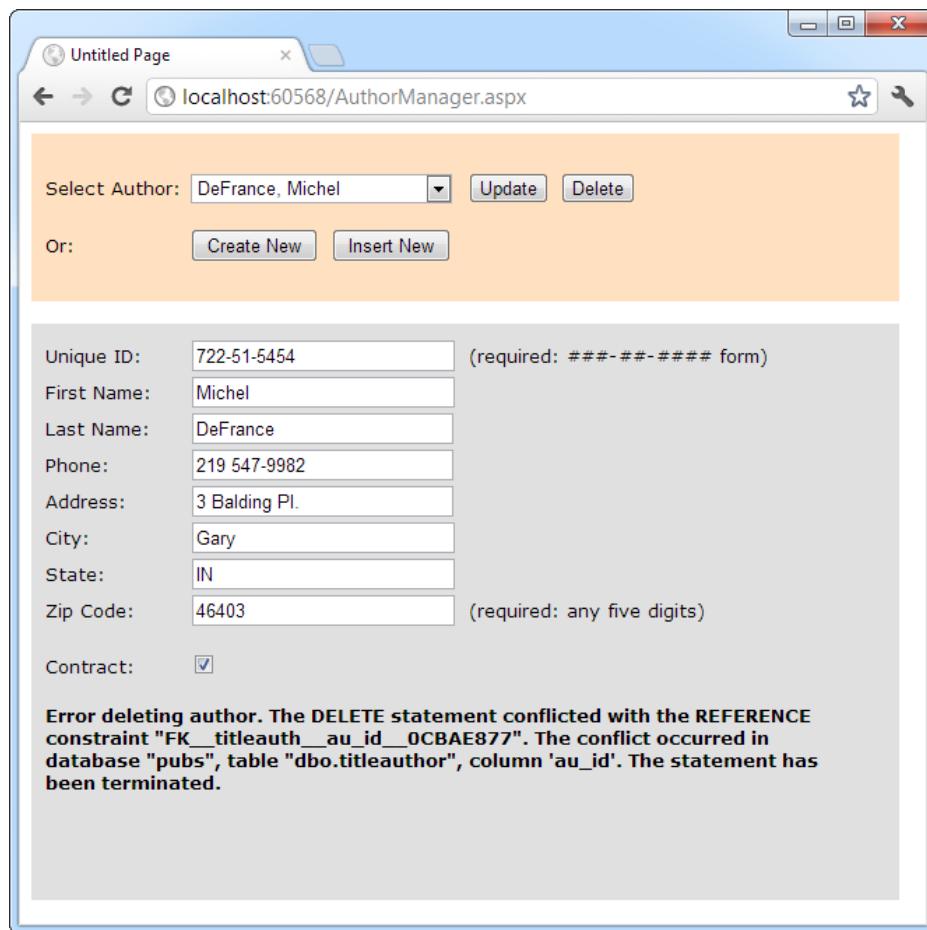


Figure 14-14. A failed delete attempt

To get around this limitation, you can use the Create New and Insert New buttons to add a new record and then delete this record. Because this new record won't be linked to any other records, its deletion will be allowed.

Note If you have a real-world application that needs to delete records, and these records might have records in a child table, there are several possible solutions. One slightly dangerous option is to configure the database to use *cascading deletes* to automatically wipe out linked records. Another option is to do the delete yourself, with additional ADO.NET code. But the best choice is usually not to delete the record at all (after all, you may need it for tracking and reporting later). Instead, use a bit column to keep track of records that should be displayed, such as a Discontinued column in a Products table or a Removed column in the Authors table. Then, when you query the table, then add a condition to your select query so that it doesn't retrieve those records. For example:

```
SELECT * FROM products
WHERE Discontinued=0
```

Using Disconnected Data Access

When you use *disconnected data access*, you use the DataSet to keep a copy of your data in memory. You connect to the database just long enough to fetch your data and dump it into the DataSet, and then you disconnect immediately.

There are a variety of good reasons to use the DataSet to hold onto data in memory. Here are a few:

- You need to do something time-consuming with the data. By dumping it into a DataSet first, you ensure that the database connection is kept open for as little time as possible.
- You want to use ASP.NET data binding to fill a web control (such as a GridView) with your data. Although you can use the DataReader, it won't work in all scenarios. The DataSet approach is more straightforward.
- You want to navigate backward and forward through your data while you're processing it. This isn't possible with the DataReader, which goes in one direction only—forward.
- You want to navigate from one table to another. Using the DataSet, you can store several tables of information. You can even define relationships that allow you to browse through them more efficiently.
- You want to save the data to a file for later use. The DataSet includes two methods—`WriteXml()` and `ReadXml()`—that allow you to dump the content to a file and convert it back to a live database object later.
- You need a convenient package to send data from one component to another. For example, in Chapter 22 you'll learn to build a database component that provides its data to a web page by using the DataSet. A DataReader wouldn't work in this scenario, because the database component would need to leave the database connection open, which is a dangerous design.
- You want to store some data so it can be used for future reChapter 23
demonstrates how you can use caching with the DataSet to achieve this result.

uPdATIng dIsConneCTed dATA

The Dataset tracks the changes you make to the records inside. This allows you to use the Dataset to records. The basic principle is simple. You fill a Dataset in the normal way, modify one or more records, then apply your update by using a DataAdapter.

However, ADO.NET's disconnected update feature makes far more sense in a desktop application than a web application. Desktop applications run for a long time, so they can efficiently store a batch of changes and perform them all at once. But in a web application, you need to commit your changes the moment they happen. Furthermore, the point at which you retrieve the data (when a page is first requested) and the point at which it's changed (during a postback) are different, which makes it very difficult to use the same DataAdapter object, and maintain the change-tracking information for the whole process.

For these reasons, ASP.NET web applications often use the Dataset to store data but rarely use it to make updates. Instead, they use direct commands to commit changes. This is the model you'll see in this book.

Selecting Disconnected Data

With disconnected data access, a copy of the data is retained in memory while your code is running. Figure 14-15 shows a model of the DataSet.

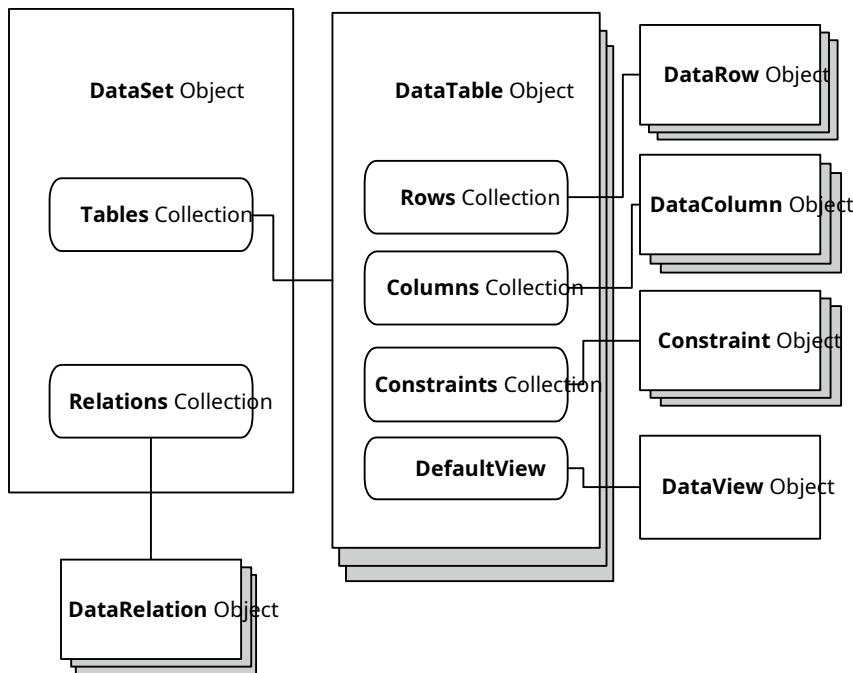


Figure 14-15. The DataSet family of objects

You fill the DataSet in much the same way that you connect a DataReader. However, although the DataReader holds a live connection, information in the DataSet is always disconnected.

The following example shows how you could rewrite the FillAuthorList() method from the earlier example to use a DataSet instead of a DataReader. The changes are highlighted in bold.

```

private void FillAuthorList()
{
    lstAuthor.Items.Clear();

    // Define ADO.NET objects.
    string selectSQL;
    selectSQL = "SELECT au_lname, au_fname, au_id FROM Authors";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataAdapter adapter = new SqlDataAdapter(cmd);
    DataSet dsPubs = new DataSet();

    // Try to open database and read information.
    try
        
```

```

{
    con.Open();

    // All the information is transferred with one command.
    // This command creates a new DataTable (named Authors)
    // inside the DataSet.
    adapter.Fill(dsPubs, "Authors");
}
catch (Exception err)
{
    lblStatus.Text = "Error reading list of names. ";
    lblStatus.Text += err.Message;
}
finally
{
    con.Close();
}

foreach (DataRow row in dsPubs.Tables["Authors"].Rows)
{
    ListItem newItem = new ListItem();
    newItem.Text = row["au_lname"] + ", " +
        row["au_fname"];
    newItem.Value = row["au_id"].ToString();
    lstAuthor.Items.Add(newItem);
}
}

```

The DataAdapter.Fill() method takes a DataSet and inserts one table of information. In this case, the table is named Authors, but any name could be used. That name is used later to access the appropriate table in the DataSet.

To access the individual DataRows, you can loop through the Rows collection of the appropriate table. Each piece of information is accessed by using the field name, as it was with the DataReader.

Selecting Multiple Tables

A DataSet can contain as many tables as you need, and you can even add relationships between the tables to better emulate the underlying relational data source. Unfortunately, you have no way to connect tables together automatically based on relationships in the underlying data source. However, you can add relations with a few extra lines of code, as shown in the next example.

In the pubs database, authors are linked to titles by using three tables. This arrangement (called a *many-to-many* relationship, shown in Figure 14-16) allows several authors to be related to one title and several titles to be related to one author. Without the intermediate TitleAuthor table, the database would be restricted to a one-to-many relationship, which would allow only a single author for each title.

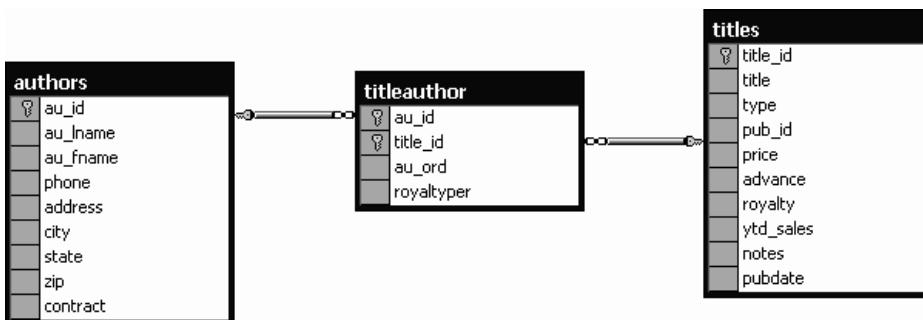


Figure 14-16. A many-to-many relationship

In an application, you would rarely need to access these tables individually. Instead, you would need to combine information from them in some way (for example, to find out what author wrote a given book). On its own, the Titles table indicates only the author ID. It doesn't provide additional information such as the author's name and address. To link this information together, you can use a special SQL Select statement called a *join query*. Alternatively, you can use the features built into ADO.NET, as demonstrated in this section.

The next example provides a simple page that lists authors and the titles they have written. The interesting thing about this page is that it's generated using ADO.NET table linking.

To start, the standard ADO.NET data access objects are created, including a DataSet. All these steps are performed in a custom CreateList() method, which is called from the Page.Load event handler so that the output is created when the page is first generated:

```

// Define the ADO.NET objects.
string connectionString =
  WebConfigurationManager.ConnectionStrings["Pubs"].ConnectionString;
SqlConnection con = new SqlConnection(connectionString);

string selectSQL = "SELECT au_lname, au_fname, au_id FROM Authors";
SqlCommand cmd = new SqlCommand(selectSQL, con);
SqlDataAdapter adapter = new SqlDataAdapter(cmd);
DataSet dsPubs = new DataSet();
  
```

Next, the information for all three tables is pulled from the database and placed in the DataSet. This task could be accomplished with three separate Command objects, but to make the code a little leaner, this example uses only one and modifies the CommandText property as needed.

```

try
{
  con.Open();
  adapter.Fill(dsPubs, "Authors");

  // This command is still linked to the data adapter.
  cmd.CommandText = "SELECT au_id, title_id FROM TitleAuthor";
  adapter.Fill(dsPubs, "TitleAuthor");

  // This command is still linked to the data adapter.
  cmd.CommandText = "SELECT title_id, title FROM Titles";
  adapter.Fill(dsPubs, "Titles");
}
  
```

```

catch (Exception err)
{
    lblList.Text = "Error reading list of names. ";
    lblList.Text += err.Message;
}
finally
{
    con.Close();
}

```

Defining Relationships

Now that all the information is in the DataSet, you can create two DataRelation objects to make it easier to navigate through the linked information. In this case, these DataRelation objects match the foreign-key restrictions that are defined in the database.

Note A *foreign key* is a constraint that you can set up in your database to link one table to another. For example, the TitleAuthor table is linked to the Titles and the Authors tables by two foreign keys. The title in the TitleAuthor table has a foreign key that binds it to the title_id field in the Titles table. similarly, the author in the TitleAuthor table has a foreign key that binds it to the au_id field in the Authors table. After these links are established, certain rules come into play. For example, you can't create a TitleAuthor record that specifies a title or author records that don't exist.

To create a DataRelation, you need to specify the linked fields from two tables, and you need to give your DataRelation a unique name. The order of the linked fields is important. The first field is the parent, and the second field is the child. (The idea here is that one parent can have many children, but each child can have only one parent. In other words, the *parent-to-child* relationship is another way of saying a *one-to-many* relationship.) In this example, each book title can have more than one entry in the TitleAuthor table. Each author can also have more than one entry in the TitleAuthor table:

```

DataRelation Titles_TitleAuthor = new DataRelation("Titles_TitleAuthor",
    dsPubs.Tables["Titles"].Columns["title_id"],
    dsPubs.Tables["TitleAuthor"].Columns["title_id"]);

```

```

DataRelation Authors_TitleAuthor = new DataRelation("Authors_TitleAuthor",
    dsPubs.Tables["Authors"].Columns["au_id"],
    dsPubs.Tables["TitleAuthor"].Columns["au_id"]);

```

After you've created these DataRelation objects, you must add them to the DataSet:

```

dsPubs.Relations.Add(Titles_TitleAuthor);
dsPubs.Relations.Add(Authors_TitleAuthor);

```

The remaining code loops through the DataSet. However, unlike the previous example, which moved through one table, this example uses the DataRelation objects to branch to the other linked tables. It works like this:

1. Select the first record from the Authors table.
2. Using the Authors_TitleAuthor relationship, find the child records that correspond to this author. To do so, you call the `DataRow.GetChildRows()` method, and pass in the `DataRelationship` object that models the relationship between the Authors and TitleAuthor table.
3. For each matching record in TitleAuthor, look up the corresponding Title record to get the full text title. To do so, you call the `DataRow.GetParentRows()` method and pass in the `DataRelationship` object that connects the TitleAuthor and Titles table.
4. Move to the next Author record and repeat the process.

The code is lean and economical:

```
foreach (DataRow rowAuthor in dsPubs.Tables["Authors"].Rows)
{
    lblList.Text += "<br /><b>" + rowAuthor["au_fname"];
    lblList.Text += " " + rowAuthor["au_lname"] + "</b><br />";

    foreach (DataRow rowTitleAuthor in
        rowAuthor.GetChildRows(Authors_TitleAuthor))
    {
        DataRow rowTitle =
            rowTitleAuthor.GetParentRows(Titles_TitleAuthor)[0];
        lblList.Text += "&nbsp;&nbsp;";
        lblList.Text += rowTitle["title"] + "<br />";
    }
}
```

Figure 14-17 shows the final result.

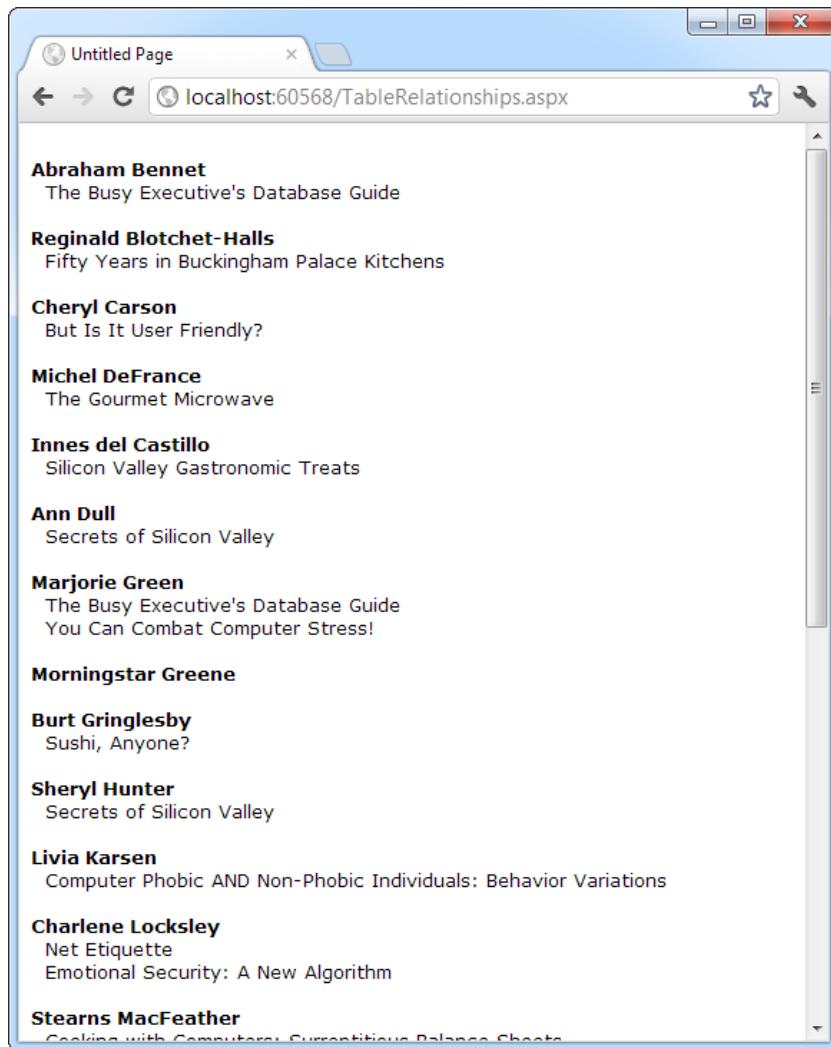


Figure 14-17. Hierarchical information from two tables

If authors and titles have a simple one-to-many relationship, you could use simpler code, as follows:

```
foreach (DataRow rowAuthor in dsPubs.Tables["Authors"].Rows)
{
    // Display author.
    foreach (DataRow rowTitle in rowAuthor.GetChildRows(Authors_Titles))
    {
        // Display title.
    }
}
```

But having seen the more complicated example, you're ready to create and manage multiple DataRelation objects on your own.

The Last Word

This chapter gave you a solid introduction to ADO.NET. You now know how to connect to a database in your web pages, retrieve the information you need, and execute commands to update, insert, and delete data.

Although you've seen all the core concepts behind ADO.NET, there's still much more to learn. In the following chapters, you'll learn about ASP.NET's data-binding system and rich data controls, and you'll see how you can use them to write more-practical data-driven web pages. And much later, you'll learn about how to take your skills to the next level by building ADO.NET-powered components (Chapter 22) and using the higher-level LINQ to Entities framework (Chapter 24).



Data Binding

In the previous chapter, you learned how to use ADO.NET to retrieve information from a database, how to store it in the DataSet, and how to apply changes by using direct commands. These techniques are flexible and powerful, but they aren't always convenient.

For example, you can use the DataSet or the DataReader to retrieve rows of information, format them individually, and add them to an HTML table on a web page. Conceptually, this isn't too difficult. However, moving through the data, formatting columns, and displaying it in the correct order still requires a lot of repetitive code. Repetitive code may be easy, but it's also error-prone, difficult to enhance, and unpleasant to read. Fortunately, ASP.NET adds a feature that allows you to skip this process and pop data directly into HTML elements and fully formatted controls. It's called *data binding*. In this chapter, you'll learn how to use data binding to display data more efficiently. You'll also learn how to use the ASP.NET *data source controls* to retrieve your data from a database without writing a line of ADO.NET code.

Introducing Data Binding

The basic principle of data binding is this: you tell a control where to find your data and how you want it displayed, and the control handles the rest of the details. Data binding in ASP.NET seems superficially similar to data binding in the world of desktop or client/server applications, but in truth, it's fundamentally different. In those environments, data binding involves creating a direct connection between a data source and a control in an application window. If the user changes a value in the onscreen control, the data in the linked database is modified automatically. Similarly, if the database changes while the user is working with it (for example, another user commits a change), the display can be refreshed automatically.

This type of data binding isn't practical in the ASP.NET world, because you can't effectively maintain a database connection over the Internet. This "direct" data binding also severely limits scalability and reduces flexibility. In fact, data binding has acquired a bad reputation for exactly these reasons.

ASP.NET data binding, on the other hand, has little in common with direct data binding. ASP.NET data binding works in one direction only. Information moves *from* a data object *into* a control. Then the data objects are thrown away, and the page is sent to the client. If the user modifies the data in a data-bound control, your program can update the corresponding record in the database, but nothing happens automatically.

ASP.NET data binding is much more flexible than old-style data binding. Many of the most powerful data-binding controls, such as the GridView and DetailsView, give you unprecedented control over the presentation of your data, allowing you to format it, change its layout, embed it in other ASP.NET controls, and so on. You'll learn about these features and ASP.NET's rich data controls in Chapter 16.

Types of ASP.NET Data Binding

Two types of ASP.NET data binding exist: single-value binding and repeated-value binding. Single-value data binding is by far the simpler of the two, whereas repeated-value binding provides the foundation for the most advanced ASP.NET data controls.

Single-Value, or “Simple,” Data Binding

You can use *single-value data binding* to add information anywhere on an ASP.NET page. You can even place information into a control property or as plain text inside an HTML tag. Single-value data binding doesn't necessarily have anything to do with ADO.NET. Instead, single-value data binding allows you to take a variable, a property, or an expression and insert it dynamically into a page. Single-value binding also helps you create templates for the rich data controls you'll study in Chapter 16.

Repeated-Value, or “List,” Binding

Repeated-value data binding allows you to display an entire table (or just a single field from a table). Unlike single-value data binding, this type of data binding requires a special control that supports it. Typically, this is a list control such as CheckBoxList or ListBox, but it can also be a much more sophisticated control such as the GridView (which is described in Chapter 16). You'll know that a control supports repeated-value data binding if it provides a DataSource property. As with single-value binding, repeated-value binding doesn't necessarily need to use data from a database, and it doesn't have to use the ADO.NET objects. For example, you can use repeated-value binding to bind data from a collection or an array.

How Data Binding Works

Data binding works a little differently depending on whether you're using single-value or repeated-value binding. To use single-value binding, you must insert a data-binding expression into the markup in the .aspx file (not the code-behind file). To use repeated-value binding, you must set one or more properties of a data control. Typically, you'll perform this initialization when the Page.Load event fires. You'll see examples of both techniques later in this chapter.

After you specify data binding, you need to activate it. You accomplish this task by calling the DataBind() method. The DataBind() method is a basic piece of functionality supplied in the Control class. It automatically binds a control and any child controls that it contains. With repeated-value binding, you can use the DataBind() method of the specific list control you're using. Alternatively, you can bind the whole page at once by calling the DataBind() method of the current Page object. After you call this method, all the data-binding expressions in the page are evaluated and replaced with the specified value.

Typically, you call the DataBind() method in the Page.Load event handler. If you forget to use it, ASP.NET will ignore your data-binding expressions, and the client will receive a page that contains empty values.

This is a general description of the whole process. To really understand what's happening, you need to work with some specific examples.

Using Single-Value Data Binding

Single-value data binding is really just a different approach to dynamic text. To use it, you add special data-binding expressions into your .aspx files. These expressions have the following format:

```
<%# expression_goes_here %>
```

This may look like a script block, but it isn't. If you try to write any code inside this tag, you will receive an error. The only thing you can add is a valid data-binding expression. For example, if you have a public or protected variable named Country in your page, you could write the following:

```
<%# Country %>
```

When you call the DataBind() method for the page, this text will be replaced with the value for Country (for example, Spain). Similarly, you could use a property or a built-in ASP.NET object as follows:

```
<%# Request.Browser.Browser %>
```

This would substitute a string with the current browser name (for example, IE). In fact, you can even call a function defined on your page, or execute a simple expression, provided it returns a result that can be converted to text and displayed on the page. Thus, the following data-binding expressions are all valid:

```
<%# GetUserName(ID) %>
<%# 1 + (2 * 20) %>
<%# "John " + "Smith" %>
```

Remember, you place these data-binding expressions in the markup portion of your .aspx file, not your code-behind file.

A Simple Data-Binding Example

This section shows a simple example of single-value data binding. The example has been stripped to the bare minimum amount of detail needed to illustrate the concept.

You start with a variable defined in your Page class, which is called TransactionCount:

```
public partial class SimpleDataBinding : System.Web.UI.Page
{
    protected int TransactionCount;

    // (Additional code omitted.)
}
```

Note that this variable must be designated as public, protected, or internal, but not private. If you make the variable private, ASP.NET will not be able to access it when it's evaluating the data-binding expression.

Now, assume that this value is set in the Page.Load event handler by using some database lookup code. For testing purposes, the example skips this step and hard-codes a value:

```
protected void Page_Load(object sender, EventArgs e)
{
    // (You could use database code here
    // to look up a value for TransactionCount.)
    TransactionCount = 10;

    // Now convert all the data-binding expressions on the page.
    this.DataBind();
}
```

Two actions take place in this event handler: the TransactionCount variable is set to 10, and all the data-binding expressions on the page are bound. Currently, no data-binding expressions exist, so this method has no effect. Notice that this example uses the this keyword to refer to the current page. You could just write DataBind() without the this keyword, because the default object is the current Page object. However, using the this keyword helps clarify which object is being used.

To make this data binding accomplish something, you need to add a data-binding expression. Usually, it's easiest to add this value directly to the markup in the .aspx file. To do so, click the Source button at the bottom of the web page designer window. Figure 15-1 shows an example with a Label control.

The screenshot shows the 'SimpleDataBinding.aspx' file in the 'Source' view of the Web Page Designer. The code includes an ASP.NET page declaration, DOCTYPE, HTML, head, and body sections. Inside the body, there is a form containing a Label control with a dynamic text value. The browser zoom level is set to 100%.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="SimpleDataBinding.aspx.cs" %>

<!DOCTYPE html>

<html>
<head runat="server">
    <title>Simple Data Binding</title>
    <link href="StyleSheet.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label id="lblDynamic" runat="server" Font-Size="X-Large" >
                There were <%# TransactionCount %> transactions today.
                I see that you are using <%# Request.Browser.Browser %>.
            </asp:Label>
        </div>
    </form>
</body>
</html>
```

Figure 15-1. Source view in the web page designer

To add your expression, find the tag for the Label control. Modify the text inside the label as shown here:

```
<asp:Label id="lblDynamic" runat="server" Font-Size="X-Large">
    There were <%# TransactionCount %> transactions today.
    I see that you are using <%# Request.Browser.Browser %>.
</asp:Label>
```

This example uses two separate data-binding expressions, which are inserted along with the normal static text. The first data-binding expression references the TransactionCount variable, and the second uses the built-in Request object to determine some information about the user's browser. When you run this page, the output looks like Figure 15-2.

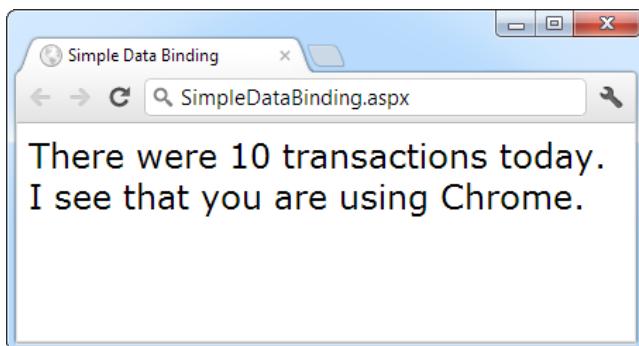


Figure 15-2. The result of data binding

The data-binding expressions have been automatically replaced with the appropriate values. If the page is posted back, you could use additional code to modify TransactionCount, and as long as you call the DataBind() method, that information will be popped into the page in the data-binding expression you've defined.

If, however, you forget to call the DataBind() method, the data-binding expressions will be ignored, and the user will see a somewhat confusing window that looks like Figure 15-3.

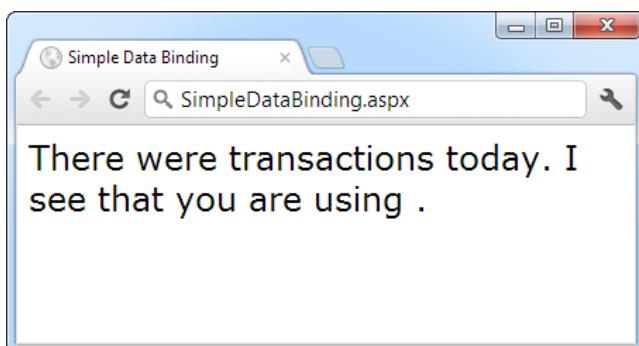


Figure 15-3. The non-data-bound page

Note When using single-value data binding, you need to consider when you should call the DataBind() method. For example, if you made the mistake of calling it before you set the TransactionCount variable, the corresponding expression would just be converted to 0. Remember, data binding is a one-way street. This means that changing the TransactionCount variable after you've used the DataBind() method won't produce any visible effect. Unless you call the DataBind() method again, the displayed value won't be updated.

Simple Data Binding with Properties

The previous example uses a data-binding expression to set static text information inside a label tag. However, you can also use single-value data binding to set other types of information on your page, including control properties. To do this, you simply have to know where to put the data-binding expression in the web page markup.

For example, consider the following page, which defines a variable named URL and uses it to point to a picture in the application directory:

```
public partial class DataBindingUrl : System.Web.UI.Page
{
    protected string URL;

    protected void Page_Load(Object sender, EventArgs e)
    {
        URL = "Images/picture.jpg";
        this.DataBind();
    }
}
```

You can now use this URL to create a label, as shown here:

```
<asp:Label id="lblDynamic" runat="server"> <%# URL %> </asp:Label>
```

You can also use it for a check box caption:

```
<asp:CheckBox id="chkDynamic" Text=" <%# URL %>" runat="server" />
```

or you can use it for a target for a hyperlink:

```
<asp:Hyperlink id="lnkDynamic" Text="Click here!" NavigateUrl="<%# URL %>" runat="server" />
```

You can even use it for a picture:

```
<asp:Image id="imgDynamic" ImageUrl=" <%# URL %>" runat="server" />
```

The only trick is that you need to edit these control tags by hand. Figure 15-4 shows what a page that uses all these elements would look like.

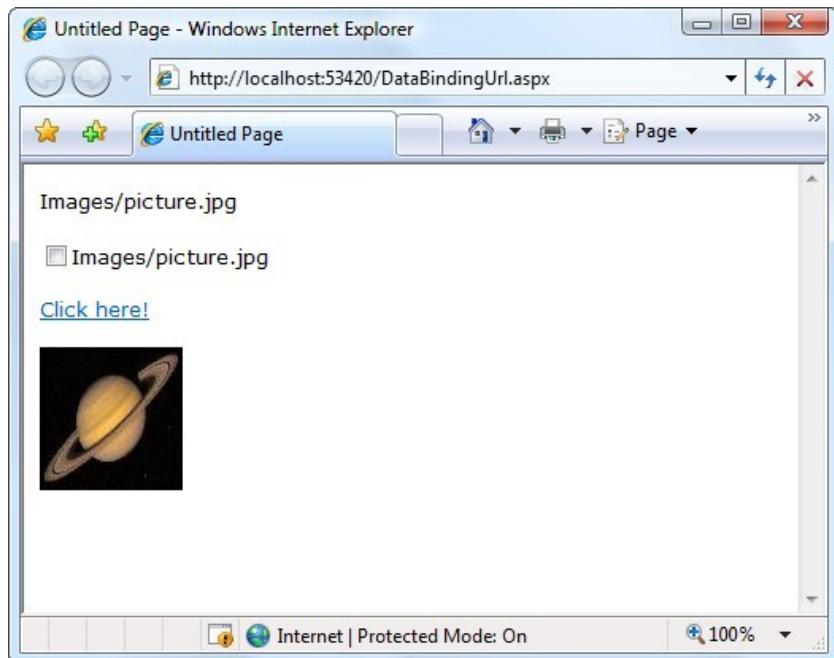


Figure 15-4. Multiple ways to bind the same data

To examine this example in more detail, try the sample code for this chapter.

Problems with Single-Value Data Binding

Before you start using single-value data-binding techniques in every aspect of your ASP.NET programs, you should consider some of the serious drawbacks that this approach can present:

Putting code into a page's user interface: One of ASP.NET's great advantages is that it allows developers to separate the user interface code (the HTML and control tags in the .aspx file) from the actual code used for data access and all other tasks (in the code-behind file). However, overenthusiastic use of single-value data binding can encourage you to disregard that distinction and start coding function calls and even operations into your page. If not carefully managed, this can lead to complete disorder.

Fragmenting code: When using data-binding expressions, it may not be obvious where the functionality resides for different operations. This is particularly a problem if you blend both approaches—for example, if you use data binding to fill a control and also modify that control directly in code. Even worse, the data-binding code may have certain dependencies that aren't immediately obvious. If the page code changes, or a variable or function is removed or renamed, the corresponding data-binding expression could stop providing valid information without any explanation or even an obvious error. All of these details make it more difficult to maintain your code, and make it more difficult for multiple developers to work together on the same project.

Of course, some developers love the flexibility of single-value data binding and use it to great effect, making the rest of their code more economical and streamlined. It's up to you to be aware of (and avoid) the potential drawbacks.

Note in one case, single-value data binding is quite useful—when building *templates*. Templates declare a block of markup that's reused for each record in a table. However, they work only with certain rich data controls such as the gridView. You'll learn more about this feature in Chapter 16.

If you decide not to use single-value data binding, you can accomplish the same thing by using code. For example, you could use the following event handler to display the same output as the first label example:

```
protected void Page_Load(Object sender, EventArgs e)
{
    TransactionCount = 10;
    lblDynamic.Text = "There were " + TransactionCount.ToString();
    lblDynamic.Text += " transactions today. ";
    lblDynamic.Text += "I see that you are using " + Request.Browser.Browser;
}
```

This code dynamically fills in the label without using data binding. The trade-off is more code.

When you use data-binding expressions, you end up complicating your markup with additional details about your code (such as the names of the variables in your code-behind class). When you use the code-only approach, you end up doing the reverse—complicating your code with additional details about the page markup (such as the text you want to display). In many cases, the best approach depends on your specific scenario. Data-binding expressions are great for injecting small bits of information into an otherwise detailed page. The dynamic code approach gives you more flexibility, and works well when you need to perform more-extensive work to shape the page (for example, interacting with multiple controls, changing content and formatting, retrieving the information you want to display from different sources, and so on).

Using Repeated-Value Data Binding

Although using simple data binding is optional, *repeated-value binding* is so useful that almost every ASP.NET application will want to use it somewhere.

Repeated-value data binding works with the ASP.NET list controls (and the rich data controls described in the next chapter). To use repeated-value binding, you link one of these controls to a data source (such as a field in a data table). When you call `DataBind()`, the control automatically creates a full list by using all the corresponding values. This saves you from writing code that loops through the array or data table and manually adds elements to a control. Repeated-value binding can also simplify your life by supporting advanced formatting and template options that automatically configure how the data should look when it's placed in the control.

To create a data expression for list binding, you need to use a list control that explicitly supports data binding. Luckily, ASP.NET provides a number of list controls, many of which you've probably already used in other applications or examples:

ListBox, DropDownList, CheckBoxList, and RadioButtonList: These web controls provide a list for a single field of information.

HtmlSelect: This server-side HTML control represents the HTML `<select>` element and works essentially the same way as the `ListBox` web control. Generally, you'll use this control only for backward compatibility.

GridView, DetailsView, FormView, and ListView: These rich web controls allow you to provide repeating lists or grids that can display more than one field of information at a time. For example, if you bind one of these controls to a full-fledged table in a DataSet, you can display the values from multiple fields. These controls offer the most powerful and flexible options for data binding.

With repeated-value data binding, you can write a data-binding expression in your .aspx file, or you can apply the data binding by setting control properties. In the case of the simpler list controls, you'll usually just set properties. Of course, you can set properties in many ways, such as by using code in a code-behind file or by modifying the control tag in the .aspx file, possibly with the help of Visual Studio's Properties window. The approach you take doesn't matter. The important detail is that with the simple list controls, you don't use any `<%# expression %>` data-binding expressions.

To continue any further with data binding, it will help to divide the subject into a few basic categories. You'll start by looking at data binding with the list controls.

Data Binding with Simple List Controls

In some ways, data binding to a list control is the simplest kind of data binding. You need to follow only three steps:

1. Create and fill some kind of data object. You have numerous options, including an array, the basic ArrayList and Hashtable collections, the strongly typed List and Dictionary collections, and the DataTable and DataSet objects. Essentially, you can use any type of collection that supports the IEnumerable interface, although you'll discover that each class has specific advantages and disadvantages.
2. Link the object to the appropriate control. To do this, you need to set only a couple of properties, including DataSource. If you're binding to a full DataSet, you'll also need to set the DataMember property to identify the appropriate table you want to use.
3. Activate the binding. As with single-value binding, you activate data binding by using the DataBind() method, either for the specific control or for all contained controls at once by using the DataBind() method for the current page.

This process is the same whether you're using the ListBox, the DropDownList, the CheckBoxList, the RadioButtonList, or even the HtmlSelect control. All these controls provide the same properties and work the same way. The only difference is in the way they appear on the final web page.

A Simple List-Binding Example

To try this type of data binding, add a ListBox control to a new web page. Then use the Page.Load event handler to create a strongly typed List collection to use as a data source:

```
List<string> fruit = new List<string>();
fruit.Add("Kiwi");
fruit.Add("Pear");
fruit.Add("Mango");
fruit.Add("Blueberry");
fruit.Add("Apricot");
fruit.Add("Banana");
fruit.Add("Peach");
fruit.Add("Plum");
```

As you learned in Chapter 3, strongly typed collections such as List are ideal when you want your collection to hold just a single type of object (for example, just strings). When you use the generic collections, you choose the item type you want to use, and the collection object is “locked in” to your choice. This means that if you try to add another type of object that doesn’t belong in the collection, you’ll get a compile-time error warning you of the mistake. And when you pull an item out of the collection, you don’t need to write casting code to convert it to the right type, because the compiler already knows what type of objects you’re using.

Now you can link this collection to the ListBox control:

```
IstItems.DataSource=fruit;
```

To activate the binding, use the DataBind() method:

```
this.DataBind();
```

You could also use IstItems.DataBind() to bind just the ListBox control. Figure 15-5 shows the resulting web page.

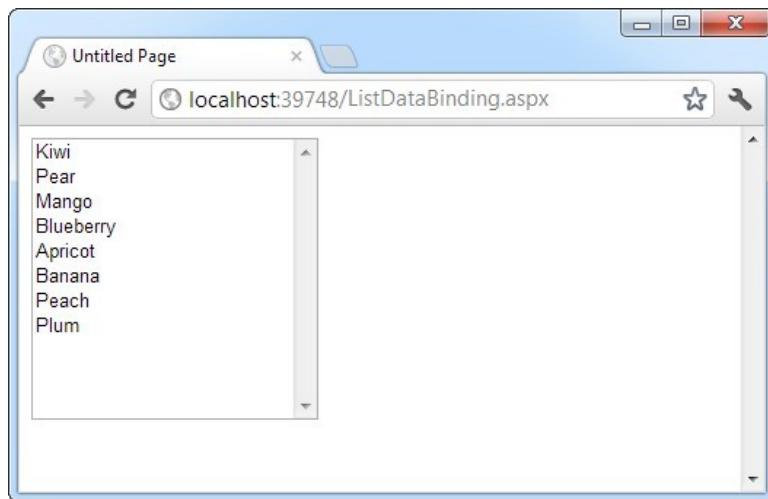


Figure 15-5. A data-bound list

This technique can save quite a few lines of code. This example doesn’t offer a lot of savings because the collection is created just before it’s displayed. In a more realistic application, however, you might be using a function that returns a ready-made collection to you:

```
List<string> fruit;
fruit = GetFruitsInSeason("Summer");
```

In this case, it’s extremely simple to add the extra two lines needed to bind and display the collection in the window:

```
IstItems.DataSource = fruit;
this.DataBind();
```

or you could even change it to the following, even more compact, code:

```
IstItems.DataSource = GetFruitsInSeason("Summer");
this.DataBind();
```

On the other hand, consider the extra trouble you would have to go through if you didn't use data binding. This type of savings compounds rapidly, especially when you start combining data binding with multiple controls, advanced objects such as DataSets, or advanced controls that apply formatting through templates.

Multiple Binding

You can bind the same data list object to multiple controls. Consider the following example, which compares all the types of list controls at your disposal by loading them with the same information:

```
protected void Page_Load(Object sender, EventArgs e)
{
    // Create and fill the collection.
    List<string> fruit = new List<string>();
    fruit.Add("Kiwi");
    fruit.Add("Pear");
    fruit.Add("Mango");
    fruit.Add("Blueberry");
    fruit.Add("Apricot");
    fruit.Add("Banana");
    fruit.Add("Peach");
    fruit.Add("Plum");

    // Define the binding for the list controls.
    MyListBox.DataSource = fruit;
    MyDropDownListBox.DataSource = fruit;
    MyHtmlSelect.DataSource = fruit;
    MyCheckBoxList.DataSource = fruit;
    MyRadioButtonList.DataSource = fruit;

    // Activate the binding.
    this.DataBind();
}
```

Figure 15-6 shows the rendered page.

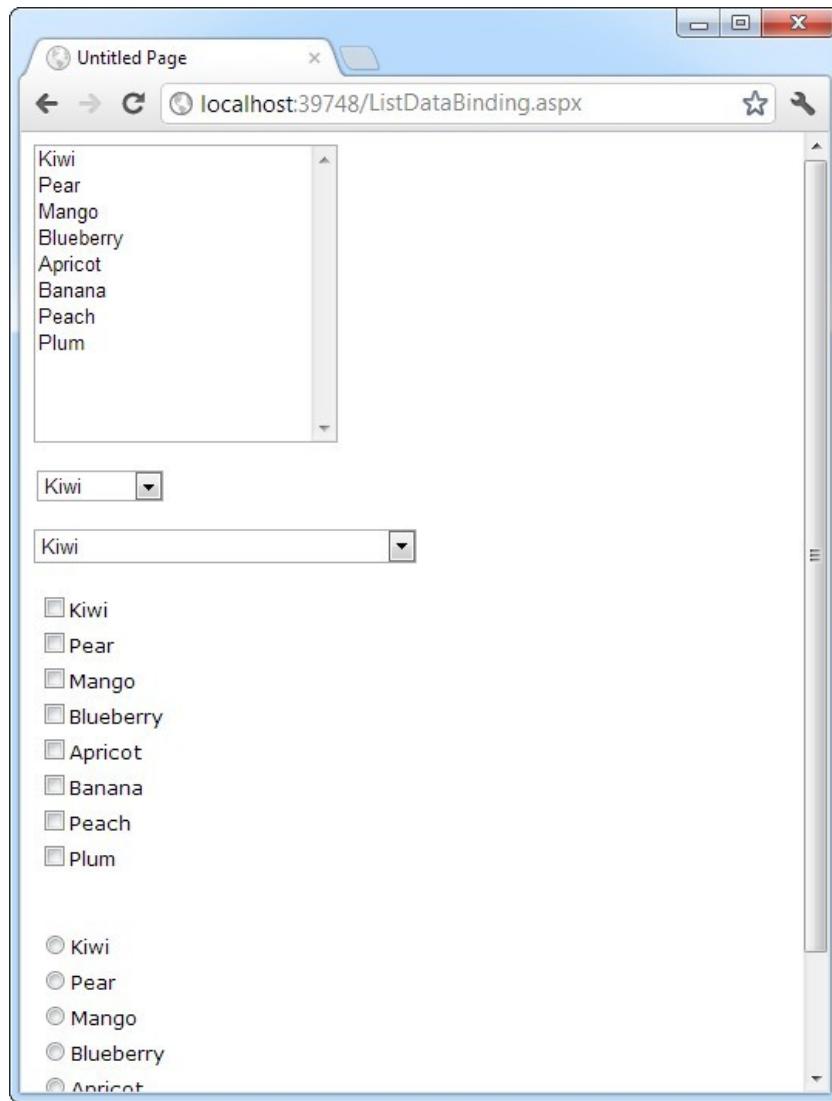


Figure 15-6. Multiple bound lists

This is another area where ASP.NET data binding may differ from what you have experienced in a desktop application. In traditional data binding, all the different controls are sometimes treated like “views” on the same data source, and you can work with only one record from the data source at a time. In this type of data binding, when you select Pear in one list control, the other list controls automatically refresh so that they too have Pear selected (or the corresponding information from the same row). This isn’t how ASP.NET uses data binding. If you want this sort of effect, you need to write custom code to pull it off.

Data Binding with a Dictionary Collection

A *dictionary collection* is a special kind of collection in which every item (or *definition*, to use the dictionary analogy) is indexed with a specific key (or *dictionary word*). This is similar to the way that built-in ASP.NET collections such as Session, Application, and Cache work.

Dictionary collections always need keys. This makes it easier to retrieve the item you want. In ordinary collections, such as the List, you need to find the item you want by its index number position, or—more often—by traveling through the whole collection until you come across the right item. With a dictionary collection, you retrieve the item you want by using its key. Generally, ordinary collections make sense when you need to work with all the items at once, while dictionary collections make sense when you frequently retrieve a single specific item.

You can use two basic dictionary-style collections in .NET. The Hashtable collection (in the System.Collections namespace) allows you to store any type of object and use any type of object for the key values. The Dictionary collection (in the System.Collections.Generic namespace) uses generics to provide the same “locking in” behavior as the List collection. You choose the item type and the key type upfront to prevent errors and reduce the amount of casting code you need to write.

The following example uses the Dictionary collection class, which it creates once—the first time the page is requested. You create a Dictionary object in much the same way you create a List collection. The only difference is that you need to supply a unique key for every item. This example uses the lazy practice of assigning a sequential number for each key:

```
protected void Page_Load(Object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        // Use integers to index each item. Each item is a string.
        Dictionary<int, string> fruit = new Dictionary<int, string>();

        fruit.Add(1, "Kiwi");
        fruit.Add(2, "Pear");
        fruit.Add(3, "Mango");
        fruit.Add(4, "Blueberry");
        fruit.Add(5, "Apricot");
        fruit.Add(6, "Banana");
        fruit.Add(7, "Peach");
        fruit.Add(8, "Plum");

        // Define the binding for the list controls.
        MyListBox.DataSource = fruit;

        // Choose what you want to display in the list.
        MyListBox.DataTextField = "Value";

        // Activate the binding.
        this.DataBind();
    }
}
```

There's one new detail here. It's this line:

```
MyListBox.DataTextField="Value";
```

Each item in a dictionary-style collection has both a key and a value associated with it. If you don't specify which property you want to display, ASP.NET simply calls the ToString() method on each collection item. This

may or may not produce the result you want. However, by inserting this line of code, you control exactly what appears in the list. The page will now appear as expected, with all the fruit names.

Note notice that you need to enclose the property name in quotation marks. ASP.nET uses reflection to your object and find the property that has the name Value at runtime.

You might want to experiment with what other types of collections you can bind to a list control. One interesting option is to use a built-in ASP.NET control such as the Session object. An item in the list will be created for every currently defined Session variable, making this trick a nice little debugging tool to quickly check current session information.

Using the DataValueField Property

Along with the DataTextField property, all list controls that support data binding also provide a DataValueField property, which adds the corresponding information to the value attribute in the control element. This allows you to store extra (undisplayed) information that you can access later. For example, you could use these two lines to define your data binding with the previous example:

```
MyListBox.DataTextField = "Value";
MyListBox.DataValueField = "Key";
```

The control will appear the same, with a list of all the fruit names in the collection. However, if you look at the rendered HTML that's sent to the client browser, you'll see that value attributes have been set with the corresponding numeric key for each item:

```
<select name="MyListBox" id="MyListBox" >
  <option value="1">Kiwi</option>
  <option value="2">Pear</option>
  <option value="3">Mango</option>
  <option value="4">Blueberry</option>
  <option value="5">Apricot</option>
  <option value="6">Banana</option>
  <option value="7">Peach</option>
  <option value="8">Plum</option>
</select>
```

You can retrieve this value later by using the SelectedItem property to get additional information. For example, you could set the AutoPostBack property of the list control to true, and add the following code:

```
protected void MyListBox_SelectedIndexChanged(Object sender,
  EventArgs e)
{
  lblMessage.Text = "You picked: " + MyListBox.SelectedItem.Text;
  lblMessage.Text += " which has the key: " + MyListBox.SelectedItem.Value;
}
```

Figure 15-7 demonstrates the result. This technique is particularly useful with a database. You could embed a unique ID into the value property and be able to quickly look up a corresponding record depending on the user's selection by examining the value of the SelectedItem object.

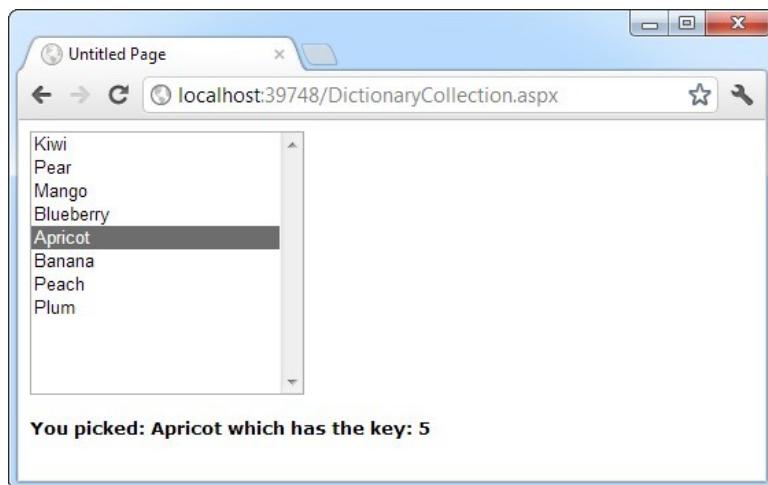


Figure 15-7. Binding to the key and value properties

Note that for this to work, you can't regenerate the list after every postback. If you do, the selected item information will be lost and an error will occur. The preceding example handles this by checking the Page. IsPostBack property. If it's false (which indicates that the page is being requested for the first time), the page builds the list. When the page is rendered, the current list of items is stored in view state. When the page is posted back, the list of items already exists and doesn't need to be re-created.

Data Binding with ADO.NET

So far, the examples in this chapter have dealt with data binding that doesn't involve databases or any part of ADO.NET. Although this is an easy way to familiarize yourself with the concepts, and a useful approach in its own right, you get the greatest advantage of data binding when you use it in conjunction with a database.

When you're using data binding with the information drawn from a database, the data-binding process takes place in the same three steps. First you create your data source, which will be a DataReader or DataSet object. A DataReader generally offers the best performance, but it limits your data binding to a single control because it is a forward-only reader. As it fills a control, it traverses the results from beginning to end. After it's finished, it can't go back to the beginning; so it can't be used in another data-binding operation. For this reason, a DataSet is a more common choice.

The next example creates a DataSet and binds it to a list. In this example, the DataSet is filled by hand, but it could just as easily be filled by using a DataAdapter object, as you saw in the previous chapter.

To fill a DataSet by hand, you need to follow several steps:

1. Create the DataSet.
2. Create a new DataTable and add it to the DataSet.Tables collection.
3. Define the structure of the table by adding DataColumn objects (one for each field) to the DataTable.Columns collection.
4. Supply the data. You can get a new, blank row that has the same structure as your DataTable by calling the DataTable.NewRow() method. You must then set the data in all its fields, and add the DataRow to the DataTable.Rows collection.

Here's how the code unfolds:

```
// Define a DataSet with a single DataTable.
DataSet dsInternal = new DataSet();
dsInternal.Tables.Add("Users");

// Define two columns for this table.
dsInternal.Tables["Users"].Columns.Add("Name");
dsInternal.Tables["Users"].Columns.Add("Country");

// Add some actual information into the table.
DataRow rowNew = dsInternal.Tables["Users"].NewRow();
rowNew["Name"] = "John";
rowNew["Country"] = "Uganda";
dsInternal.Tables["Users"].Rows.Add(rowNew);

rowNew = dsInternal.Tables["Users"].NewRow();
rowNew["Name"] = "Samantha";
rowNew["Country"] = "Belgium";
dsInternal.Tables["Users"].Rows.Add(rowNew);

rowNew = dsInternal.Tables["Users"].NewRow();
rowNew["Name"] = "Rico";
rowNew["Country"] = "Japan";
dsInternal.Tables["Users"].Rows.Add(rowNew);
```

Next you bind the DataTable from the DataSet to the appropriate control. Because list controls can show only a single column at a time, you also need to choose the field you want to display for each item by setting the DataTextField property:

```
// Define the binding.
lstUser.DataSource = dsInternal.Tables["Users"];
lstUser.DataTextField = "Name";
```

Alternatively, you could use the entire DataSet for the data source, instead of just the appropriate table. In that case, you would have to select a table by setting the control'sDataMember property. This is an equivalent approach, but the code is slightly different:

```
// Define the binding.
lstUser.DataSource = dsInternal;
lstUser.DataMember = "Users";
lstUser.DataTextField = "Name";
```

As always, the last step is to activate the binding:

```
this.DataBind();
```

The final result is a list with the information from the specified database field, as shown in Figure 15-8. The list box will have an entry for every single record in the table, even if it appears more than once, from the first row to the last.

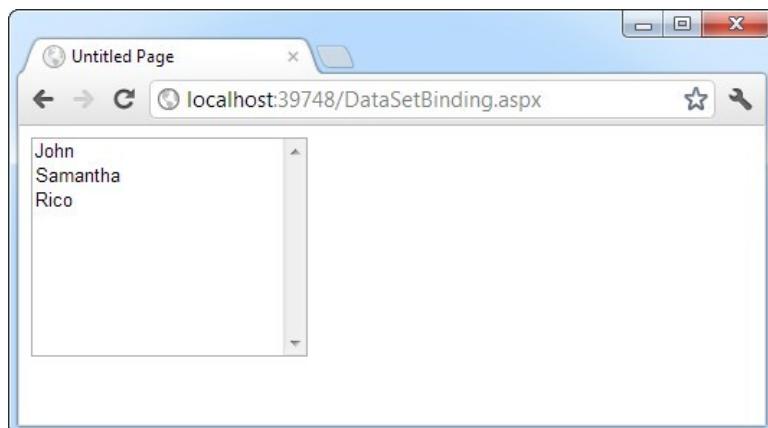


Figure 15-8. DataSet binding

Tip The simple list controls require you to bind their Text or Value property to a single data field in the source object. However, much more flexibility is provided by the more advanced data-binding controls in the next chapter. They allow fields to be combined in just about any way you can imagine.

Creating a Record Editor

The next example is more practical. It's a good illustration of how you might use data binding in a full ASP.NET application. This example allows the user to select a record and update one piece of information by using data-bound list controls.

The first step is to add the connection string to your web.config file. This example uses the Products table from the Northwind database included with many versions of SQL Server. Here's how you can define the connection string for SQL Server Express LocalDB:

```
<configuration>
  <connectionStrings>
    <add name="Northwind" connectionString=
      "Data Source=(localdb)\v11.0;Initial Catalog=Northwind;Integrated
      Security=SSPI" />
  </connectionStrings>
  ...
</configuration>
```

To use the full version of SQL Server, remove the (localdb)\v11.0 portion. To use a database server on another computer, supply the computer name for the Data Source connection string property. (For more details about connection strings, refer to Chapter 14.)

The next step is to retrieve the connection string and store it in a private variable in the Page class so that every part of your page code can access it easily. After you've imported the System.Web.Configuration namespace, you can create a member variable in your code-behind class that's defined like this:

```
private string connectionString =
  WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
```

The next step is to create a drop-down list that allows the user to choose a product for editing. The PageLoad event handler takes care of this task—retrieving the data, binding it to the drop-down list control, and then activating the binding. Before you go any further, make sure you've imported the System.Data.SqlClient namespace, which allows you to use the SQL Server provider to retrieve data.

```
protected void Page_Load(Object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        // Define the ADO.NET objects for selecting products from the database.
        string selectSQL = "SELECT ProductName, ProductID FROM Products";
        SqlConnection con = new SqlConnection(connectionString);
        SqlCommand cmd = new SqlCommand(selectSQL, con);

        // Open the connection.
        con.Open();

        // Define the binding.
        lstProduct.DataSource = cmd.ExecuteReader();
        lstProduct.DataTextField = "ProductName";
        lstProduct.DataValueField = "ProductID";

        // Activate the binding.
        this.DataBind();

        con.Close();

        // Make sure nothing is currently selected in the list box.
        lstProduct.SelectedIndex = -1;
    }
}
```

Once again, the list is filled only the first time the page is requested (and stored in view state automatically). If the page is posted back, the list keeps its current entries. This reduces the amount of database work, and keeps the page working quickly and efficiently. You should also note that this page doesn't attempt to deal with errors. If you were using it in a real application, you'd need to use the exception-handling approach demonstrated in Chapter 14.

The actual database code is similar to what was used in the previous chapter. The example uses a Select statement but carefully limits the returned information to just the ProductName and ProductID fields, which are the only pieces of information it will use. The resulting window lists all the products defined in the database, as shown in Figure 15-9.

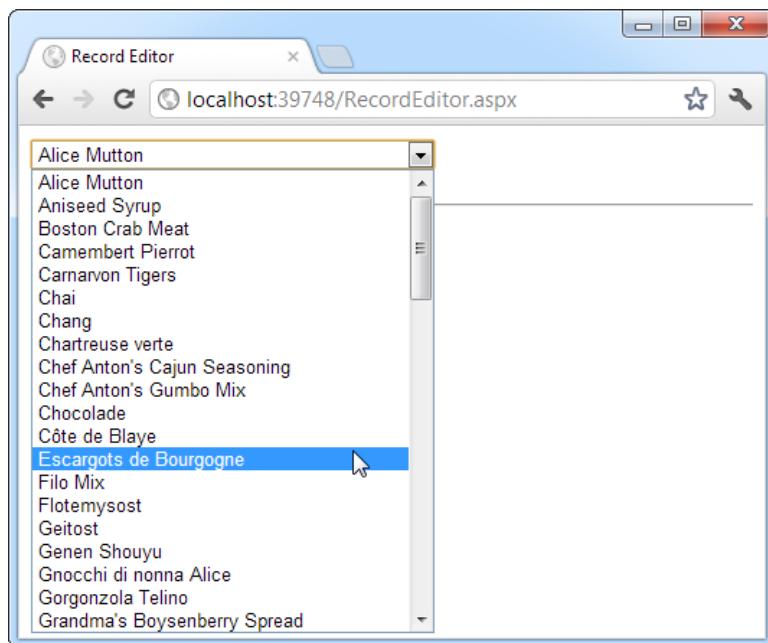


Figure 15-9. Product choices

The drop-down list enables AutoPostBack, so as soon as the user makes a selection, a `IstProduct_SelectedIndexChanged` event fires. At this point, your code performs the following tasks:

- It reads the corresponding record from the Products table and displays additional information about it in a label. In this case, a Join query links information from the Products and Categories tables. The code also determines what the category is for the current product. This is the piece of information it will allow the user to change.
- It reads the full list of CategoryNames from the Categories table and binds this information to a different list control. Initially, this list is hidden in a panel with its `Visible` property set to false. The code reveals the content of this panel by setting `Visible` to true.
- It highlights the row in the category list that corresponds to the current product. For example, if the current product is a Seafood category, the Seafood entry in the list box will be selected.

This logic appears fairly involved, but it's really just an application of what you've learned over the past two chapters. The full listing is as follows:

```
protected void IstProduct_SelectedIndexChanged(object sender, EventArgs e)
{
    // Create a command for selecting the matching product record.
    string selectProduct = "SELECT ProductName, QuantityPerUnit, " +
        "CategoryName FROM Products INNER JOIN Categories ON " +
        "Categories.CategoryID=Products.CategoryID " +
        "WHERE ProductID=@ProductID";
```

```

// Create the Connection and Command objects.
SqlConnection con = new SqlConnection(connectionString);
SqlCommand cmdProducts = new SqlCommand(selectProduct, con);
cmdProducts.Parameters.AddWithValue("@ProductID",
    lstProduct.SelectedItem.Value);

// Retrieve the information for the selected product.
using (con)
{
    con.Open();
    SqlDataReader reader = cmdProducts.ExecuteReader();
    reader.Read();

    // Update the display.
    lblRecordInfo.Text = "<b>Product:</b> " +
        reader["ProductName"] + "<br />";
    lblRecordInfo.Text += "<b>Quantity:</b> " +
        reader["QuantityPerUnit"] + "<br />";
    lblRecordInfo.Text += "<b>Category:</b> " + reader["CategoryName"];

    // Store the corresponding CategoryName for future reference.
    string matchCategory = reader["CategoryName"].ToString();

    // Close the reader.
    reader.Close();

    // Create a new Command for selecting categories.
    string selectCategory = "SELECT CategoryName, " +
        "CategoryID FROM Categories";
    SqlCommand cmdCategories = new SqlCommand(selectCategory, con);

    // Retrieve the category information and bind it.
    lstCategory.DataSource = cmdCategories.ExecuteReader();
    lstCategory.DataTextField = "CategoryName";
    lstCategory.DataValueField = "CategoryID";
    lstCategory.DataBind();

    // Highlight the matching category in the list.
    lstCategory.Items.FindByText(matchCategory).Selected = true;
}

pnlCategory.Visible = true;
}

```

You could improve this code in several ways. It probably makes the most sense to remove these data access routines from this event handler and put them into more-generic functions. For example, you could use a function that accepts a ProductID and returns a single DataRow with the associated product information. Another improvement would be to use a stored procedure to retrieve this information.

The end result is a window that updates itself dynamically whenever a new product is selected, as shown in Figure 15-10.

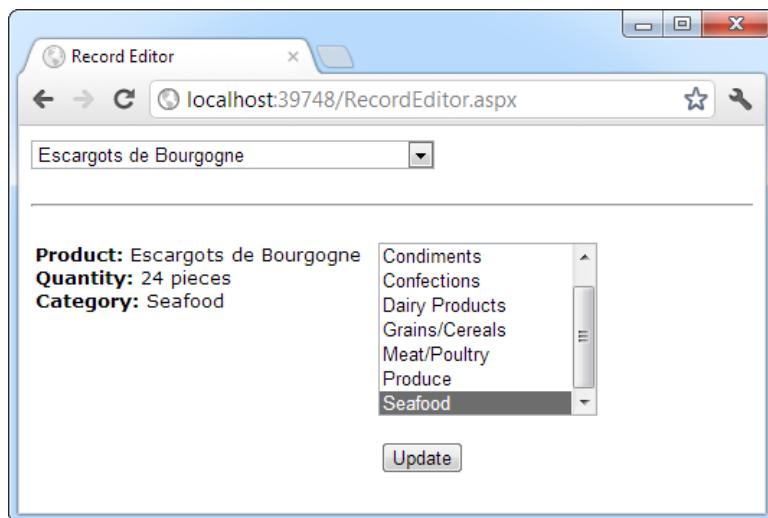


Figure 15-10. Product information

This example still has one more trick in store. If the user selects a different category and clicks Update, the change is made in the database. Of course, this means creating new Connection and Command objects, as follows:

```
protected void cmdUpdate_Click(object sender, EventArgs e)
{
    // Define the Command.
    string updateCommand = "UPDATE Products " +
        "SET CategoryID=@CategoryID WHERE ProductID=@ProductID";

    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(updateCommand, con);

    cmd.Parameters.AddWithValue("@CategoryID", lstCategory.SelectedItem.Value);
    cmd.Parameters.AddWithValue("@ProductID", lstProduct.SelectedItem.Value);

    // Perform the update.
    using (con)
    {
        con.Open();
        cmd.ExecuteNonQuery();
    }
}
```

You could easily extend this example so that it allows you to edit all the properties in a product record. But before you try that, you might want to experiment with the rich data controls that are shown in the next chapter. Using these controls, you can create sophisticated lists and grids that provide automatic features for selecting, editing, and deleting records.

Working with Data Source Controls

In Chapter 14, you saw how to directly connect to a database, execute a query, loop through the records in the result set, and display them on a page. In this chapter, you've already seen a simpler option—with data binding, you can write your data access logic and then show the results in the page with no looping or control manipulation required. Now it's time to introduce *another* convenience: data source controls. Amazingly enough, data source controls allow you to create data-bound pages without writing any data access code at all.

Note As you'll soon see, often a gap exists between what you *can* do and what you *should* do. In most professional applications, you'll need to write and fine-tune your data access code for optimum performance or specific features. That's why you've spent so much time learning how ADO.NET works, rather than jumping to the data source controls.

The data source controls include any control that implements the `IDataSource` interface. The .NET Framework includes the following data source controls:

SqlDataSource: This data source allows you to connect to any data source that has an ADO.NET data provider. This includes SQL Server, Oracle, and OLE DB or ODBC data sources. When using this data source, you don't need to write the data access code.

AccessDataSource: This data source allows you to read and write the data in an Access database file (.mdb). However, its use is discouraged, because Access doesn't scale well to large numbers of users (unlike SQL Server Express).

Note Access databases do not have a dedicated server engine (such as SQL Server) that coordinates transactions of multiple people and ensures that data won't be lost or corrupted. For that reason, Access databases are suited for very small websites, where few people need to manipulate data at the same time. A much better scale data solution is SQL Server Express, which is described in Chapter 14.

ObjectDataSource: This data source allows you to connect to a custom data access class. This is the preferred approach for large-scale professional web applications, but it forces you to write much more code. You'll study the ObjectDataSource in Chapter 22.

XmlDataSource: This data source allows you to connect to an XML file. You'll learn more about XML in Chapter 18.

SiteMapDataSource: This data source allows you to connect to a .sitemap file that describes the navigational structure of your website. You saw this data source in Chapter 13.

EntityDataSource: This data source allows you to query a database by using the LINQ to Entities feature, which you'll tackle in Chapter 24.

LinqDataSource: This data source allows you to query a database by using the LINQ to SQL feature, which is a similar (but somewhat less powerful) predecessor to LINQ to Entities.

You can find all the data source controls in the Data tab of the Toolbox in Visual Studio, with the exception of the AccessDataSource.

When you drop a data source control onto your web page, it shows up as a gray box in Visual Studio. However, this box won't appear when you run your web application and request the page (see Figure 15-11).

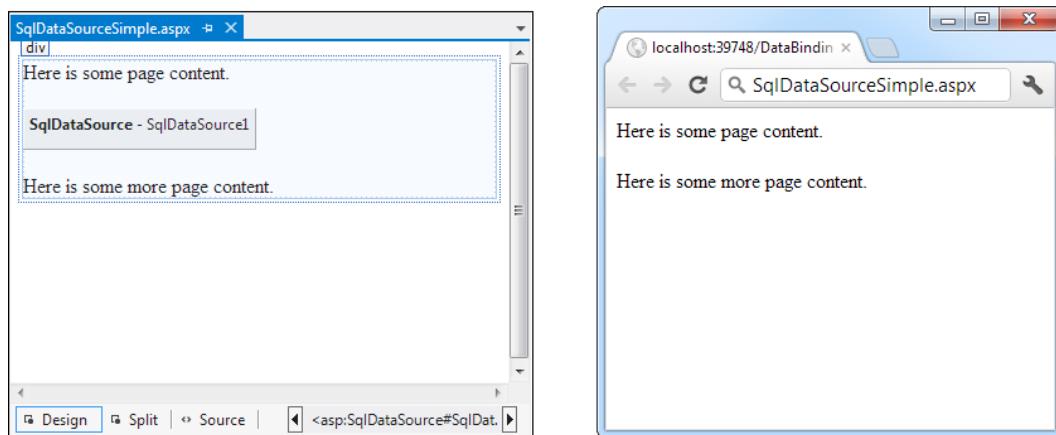


Figure 15-11. A data source control at design time and runtime

If you perform more than one data access task in the same page (for example, you need to be able to query two different tables), you'll need more than one data source control.

The Page Life Cycle with Data Binding

Data source controls can perform two key tasks:

- They can retrieve data from a data source and supply it to bound controls. When you use this feature, your bound controls are automatically filled with data. You don't even need to call DataBind().
- They can update the data source when edits take place. In order to use this feature, you must use one of ASP.NET's rich data controls, such as the GridView or DetailsView. For example, if you make an edit in the GridView and click Update, the GridView will trigger the update in the data source control, and the data source control will then update the database.

Before you can use the data source controls, you need to understand the page life cycle. The following steps explain the sequence of stages your page goes through in its lifetime. The two steps in bold (4 and 6) indicate the points where the data source controls will spring into action:

1. The page object is created (based on the .aspx file).
2. The page life cycle begins, and the Page.Init and Page.Load events fire.
3. All other control events fire.

4. If the user is applying a change, the data source controls perform their update operations now. If a row is being updated, the Updating and Updated events fire. If a row is being inserted, the Inserting and Inserted events fire. If a row is being deleted, the Deleting and Deleted events fire.
5. The Page.PreRender event fires.
6. The data source controls perform their queries and insert the data they retrieve into the bound controls. This step happens the first time your page is requested and every time the page is posted back, ensuring that you always have the most up-to-date data. The Selecting and Selected events fire at this point.
7. The page is rendered and disposed.

In the rest of this chapter, you'll take a closer look at the SqlDataSource control, and you'll use it to build the record editor example demonstrated earlier—with a lot less code.

The SqlDataSource

Data source controls turn up in the .aspx markup portion of your web page like ordinary controls. Here's an example:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server" ... />
```

The SqlDataSource represents a database connection that uses an ADO.NET provider. However, this has a catch. The SqlDataSource needs a generic way to create the Connection, Command, and DataReader objects it requires. This is possible only if your data provider includes something called a *data provider factory*. The factory has the responsibility of creating the provider-specific objects that the SqlDataSource needs to access the data source. Fortunately, .NET includes a data provider factory for each of its four data providers:

- System.Data.SqlClient
- System.Data.OracleClient
- System.Data.OleDb
- System.Data.Odbc

You can use all of these providers with the SqlDataSource. You choose your data source by setting the provider name. Here's a SqlDataSource that connects to a SQL Server database by using the SQL Server provider:

```
<asp:SqlDataSource ProviderName="System.Data.SqlClient" ... />
```

Technically, you can omit this piece of information, because the System.Data.SqlClient provider factory is the default.

Note if you have an up-to-date third-party provider (such as ODP.nET for accessing Oracle databases) also include a provider factory that allows you to use it with the SqlDataSource.

The next step is to supply the required connection string—without it, you cannot make any connections. Although you can hard-code the connection string directly in the SqlDataSource tag, keeping it in the <connectionStrings> section of the web.config file is always better, to guarantee greater flexibility and ensure that you won't inadvertently change the connection string.

To refer to a connection string in your .aspx markup, you use a special syntax in this format:

```
<%$ ConnectionStrings:[NameOfConnectionString] %>
```

This looks like a data-binding expression, but it's slightly different. (For one thing, it begins with the character sequence <%\$ instead of <%#.)

For example, if you have a connection string named Northwind in your web.config file that looks like this:

```
<configuration>
  <connectionStrings>
    <add name="Northwind" connectionString=
      "Data Source=(localdb)\v11.0;Initial Catalog=Northwind;Integrated
      Security=SSPI" />
  </connectionStrings>
  ...
</configuration>
```

you would specify it in the SqlDataSource by using this syntax:

```
<asp:SqlDataSource ConnectionString="<%$ ConnectionStrings:Northwind %>" ... />
```

After you've specified the provider name and connection string, the next step is to add the query logic that the SqlDataSource will use when it connects to the database.

Tip If you want some help creating your connection string, select the SqlDataSource, open the Properties window, and select the ConnectionString property. A drop-down arrow will appear at the right side of the click that drop-down arrow, you'll see a list of all the connection strings in your web.config file. You can pick these connections, or you can choose new Connection (at the bottom of the list) to open the Add Connection box, where you can pick the database you want. Best of all, if you create a new connection, Visual Studio adds the connection string into your web.config file, so you can reuse it with other SqlDataSource objects.

Selecting Records

You can use each SqlDataSource control you create to retrieve a single query. Optionally, you can also add corresponding commands for deleting, inserting, and updating rows. For example, one SqlDataSource is enough to query and update the Customers table in the Northwind database. However, if you need to independently retrieve or update Customers and Orders information, you'll need two SqlDataSource controls.

The SqlDataSource command logic is supplied through four properties—SelectCommand, InsertCommand, UpdateCommand, and DeleteCommand—each of which takes a string. The string you supply can be inline SQL (in which case the corresponding SelectCommandType, InsertCommandType, UpdateCommandType, or DeleteCommandType property should be Text, the default) or the name of a stored procedure (in which case the command type is StoredProcedure). You need to define commands only for the types of actions you want to perform. In other words, if you're using a data source for read-only access to a set of records, you need to define only the SelectCommand property.

Note If you configure a command in the Properties window, you'll see a property named SelectQuery. SelectCommand is a virtual property that's displayed as a design-time convenience. When you click the ellipsis next to the property name, you can use a special designer to write the command text (the SelectCommand) and add the command parameters (the SelectParameters) at the same time. However, this tool works best after you've reviewed the examples in this section, and you understand the SelectCommand and SelectParameters properties really work.

Here's a complete SqlDataSource that defines a Select command for retrieving product information from the Products table:

```
<asp:SqlDataSource ID="sourceProducts" runat="server"
    ConnectionString="<%$ ConnectionStrings:Northwind %>" 
    SelectCommand="SELECT ProductName, ProductID FROM Products"
/>
```

Tip You can write the data source logic by hand, or you can use a design-time wizard that lets you create the connection and create the command logic in a graphical query builder. You can even test the query as you type to make sure it returns the records you want. To launch this tool, select the data source control on the design surface of your page, and click the Configure Data Source link in the smart tag.

This is enough to build the first stage of the record editor example shown earlier—namely, the drop-down list box that shows all the products. All you need to do is set the DataSourceID property to point to the SqlDataSource you've created. The easiest way to do this is by using the Properties window, which provides a drop-down list of all the data sources on your current web page. At the same time, make sure you set the DataTextField and DataValueField properties. After you make these changes, you'll wind up with a control tag like this:

```
<asp:DropDownList ID="lstProduct" runat="server" AutoPostBack="True"
    DataSourceID="sourceProducts" DataTextField="ProductName"
    DataValueField="ProductID" />
```

The best part about this example is that you don't need to write any code. When you run the page, the DropDownList control asks the SqlDataSource for the data it needs. At this point, the SqlDataSource executes the query you defined, fetches the information, and binds it to the DropDownList. The whole process unfolds automatically.

How the Data Source Controls Work

As you learned earlier in this chapter, you can bind to a DataReader or a DataSet. So it's worth asking—which approach does the SqlDataSource control use? It's actually your choice, depending on whether you set the DataSourceMode to SqlDataSourceMode.DataSet (the default) or to SqlDataSourceMode.DataReader. The DataSet mode is almost always better, because it supports advanced sorting, filtering, and caching settings that depend on the DataSet. All these features are disabled in DataReader mode. However, you can use the DataReader mode with extremely large grids, because it's more memory-efficient. That's because the DataReader holds only one record in memory at a time—just long enough to copy the record's information to the linked control.

Another important fact to understand about the data source controls is that when you bind more than one control to the same data source, you cause the query to be executed multiple times. For example, if two controls are bound to the same data source, the data source control performs its query twice—once for each control. This is somewhat inefficient—after all, if you wrote the data-binding code yourself by hand, you'd probably choose to perform the query once and then bind the returned DataSet twice. Fortunately, this design isn't quite as bad as it might seem.

First, you can avoid this multiple-query overhead by using caching, which allows you to store the retrieved data in a temporary memory location where it will be reused automatically. The `SqlDataSource` supports automatic caching if you set `EnableCaching` to true. Chapter 23 provides a full discussion of how caching works and how you can use it with the `SqlDataSource`.

Second, contrary to what you might expect, most of the time you won't be binding more than one control to a data source. That's because the rich data controls you'll learn about in Chapter 16—the `GridView`, `DetailsView`, and `FormView`—have the ability to present multiple pieces of data in a flexible layout. If you use these controls, you'll need to bind only one control, which allows you to steer clear of this limitation.

It's also important to remember that data binding is performed at the end of your web page processing, just before the page is rendered. This means the `Page.Load` event will fire, followed by any control events, followed by the `Page.PreRender` event. Only then will the data binding take place.

Parameterized Commands

In the previous example (which used the `SqlDataSource` to retrieve a list of products), the complete query was hard-coded. Often you won't have this flexibility. Instead, you'll want to retrieve a subset of data, such as all the products in a given category or all the employees in a specific city.

The record editor that you considered earlier offers an ideal example. After you select a product, you want to execute another command to get the full details for that product. (You might just as easily execute another command to get records that are related to this product.) To make this work, you need two data sources. You've already created the first `SqlDataSource`, which fetches limited information about every product. Here's the second `SqlDataSource`, which gets more-extensive information about a single product (the following query is split over several lines to fit the printed page):

```
<asp:SqlDataSource ID="sourceProductDetails" runat="server"
  ProviderName="System.Data.SqlClient"
  ConnectionString="<%$ ConnectionStrings:Northwind %>" 
  SelectCommand="SELECT * FROM Products WHERE ProductID=@ProductID"
/>
```

But this example has a problem. It defines a parameter (`@ProductID`) that identifies the ID of the product you want to retrieve. How do you fill in this piece of information? It turns out you need to add a `<SelectParameters>` section to the `SqlDataSource` tag. Inside this section, you must define each parameter that's referenced by your `SelectCommand` and tell the `SqlDataSource` where to find the value it should use. You do that by *mapping* the parameter to a value in a control.

Here's the corrected command:

```
<asp:SqlDataSource ID="sourceProductDetails" runat="server"
  ProviderName="System.Data.SqlClient"
  ConnectionString="<%$ ConnectionStrings:Northwind %>" 
  SelectCommand="SELECT * FROM Products WHERE ProductID=@ProductID"
  <SelectParameters>
    <asp:ControlParameter ControlID="lstProduct" Name="ProductID"
      PropertyName="SelectedValue" />
  </SelectParameters>
</asp:SqlDataSource>
```

You always indicate parameters with an @ symbol, as in @City. You can define as many parameters as you want, but you must map each one to a value by using a separate element in the SelectParameters collection. In this example, the value for the @ProductID parameter comes from the lstProduct.SelectedValue property. In other words, you are binding a value that's currently in a control to place it into a database command. (You could also use the SelectedText property to get the currently displayed text, which is the ProductName in this example.)

Now all you need to do is bind the SqlDataSource to the remaining controls where you want to display information. This is where the example takes a slightly different turn. In the previous version of the record editor, you took the information and used a combination of values to fill in details in a label and a list control. This type of approach doesn't work well with data source controls. First, you can bind only a single data field to most simple controls such as lists. Second, each bound control makes a separate request to the SqlDataSource, triggering a separate database query. This means that if you bind a dozen controls, you'll perform the same query a dozen times, with terrible performance. You can alleviate this problem with data source caching (see Chapter 23), but that would indicate you aren't designing your application in a way that lends itself well to the data source control model.

The solution is to use one of the rich data controls, such as the GridView, DetailsView, or FormView. These controls have the smarts to show multiple fields at once, in a highly flexible layout. You'll learn about these three controls in detail in the next chapter, but the following example shows a simple demonstration of how to use the DetailsView.

The DetailsView is a rich data control that's designed to show multiple fields in a data source. As long as its AutoGenerateRows is true (the default), it creates a separate row for each field, with the field caption and value. Figure 15-12 shows the result.

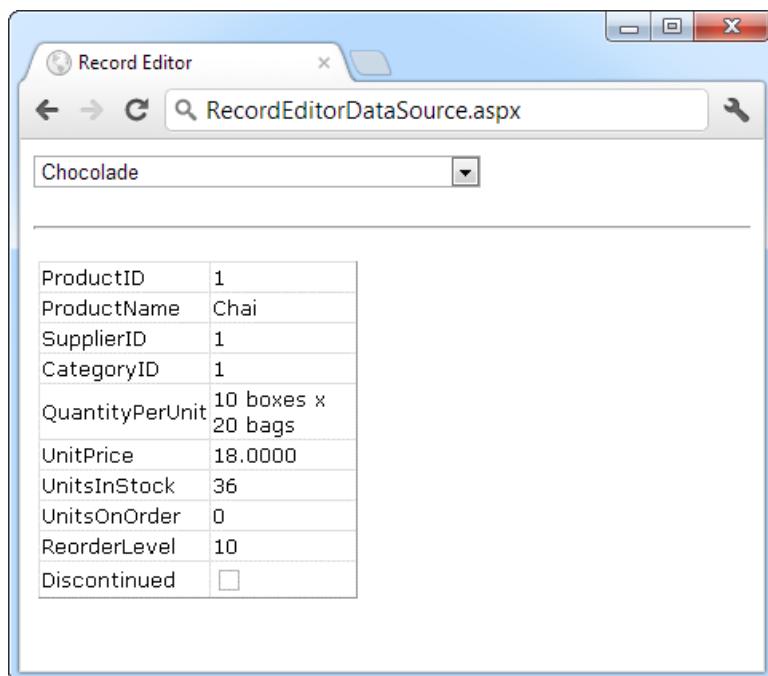


Figure 15-12. Displaying full product information in a DetailsView

Here's the basic DetailsView tag that makes this possible:

```
<asp:DetailsView ID="detailsProduct" runat="server"
    DataSourceID="sourceProductDetails" />
```

As you can see, the only property you need to set is `DataSourceID`. That binds the `DetailsView` to the `SqlDataSource` you created earlier. This `SqlDataSource` gets the full product information for a single row, based on the selection in the list control. Best of all, this whole example still hasn't required a line of code.

Other Types of Parameters

In the previous example, the `@ProductID` parameter in the second `SqlDataSource` is configured based on the selection in a drop-down list. This type of parameter, which links to a property in another control, is called a *control parameter*. But parameter values aren't necessarily drawn from other controls. You can map a parameter to any of the parameter types defined in Table 15-1.

Table 15-1. Parameter Types

Source	Control Tag	Description
Control property	<code><asp:ControlParameter></code>	A property from another control on the page.
Query string value	<code><asp:QueryStringParameter></code>	A value from the current query string.
Session state value	<code><asp:SessionParameter></code>	A value stored in the current user's session.
Cookie value	<code><asp:CookieParameter></code>	A value from any cookie attached to the current request.
Profile value	<code><asp:ProfileParameter></code>	A value from the current user's profile (see Chapter 21 for more about profiles).
Routed URL value	<code><asp:RouteParameter></code>	A value from a routed URL. Routed URLs are an advanced technique that lets you map any URL to a different page (so a request such as http://www.mysite.com/products/112 redirects to the page www.mysite.com/productdetails.aspx?id=112 , for example). To learn more about URL routing, refer to the Visual Studio Help or <i>Pro ASP.NET 4.5 in C#</i> (Apress).
A form variable	<code><asp:FormParameter></code>	A value posted to the page from an input control. Usually, you'll use a control property instead, but you might need to grab a value straight from the Forms collection if you've disabled view state for the corresponding control.

For example, you could split the earlier example into two pages. In the first page, define a list control that shows all the available products:

```
<asp:SqlDataSource ID="sourceProducts" runat="server"
  ProviderName="System.Data.SqlClient"
  ConnectionString="<%$ ConnectionStrings:Northwind %>"
  SelectCommand="SELECT ProductName, ProductID FROM Products"
/>
<asp:DropDownList ID="lstProduct" runat="server" AutoPostBack="True"
  DataSourceID="sourceProducts" DataTextField="ProductName"
  DataValueField="ProductID" />
```

Now you'll need a little extra code to copy the selected product to the query string and redirect the page. Here's a button that does just that:

```
protected void cmdGo_Click(object sender, EventArgs e)
{
    if (lstProduct.SelectedIndex != -1)
    {
        Response.Redirect(
            "QueryParameter2.aspx?prodID=" + lstProduct.SelectedValue);
    }
}
```

Finally, the second page can bind the DetailsView according to the ProductID value that's supplied in the query string:

```
<asp:SqlDataSource ID="sourceProductDetails" runat="server"
    ProviderName="System.Data.SqlClient"
    ConnectionString="<%$ ConnectionStrings:Northwind %>"
    SelectCommand="SELECT * FROM Products WHERE ProductID=@ProductID">
    <SelectParameters>
        <asp:QueryStringParameter Name="ProductID" QueryStringField="prodID" />
    </SelectParameters>
</asp:SqlDataSource>

<asp:DetailsView ID="detailsProduct" runat="server"
    DataSourceID="sourceProductDetails" />
```

Setting Parameter Values in Code

Sometimes you'll need to set a parameter with a value that isn't represented by any of the parameter classes in Table 15-1. Or you might want to manually modify a parameter value before using it. In both of these scenarios, you need to use code to set the parameter value just before the database operation takes place.

For example, consider the page shown in Figure 15-13. It includes two data-bound controls. The first is a list of all the customers in the database. Here's the markup that defines the list and its data source:

```
<asp:SqlDataSource ID="sourceCustomers" runat="server"
    ProviderName="System.Data.SqlClient"
    ConnectionString="<%$ ConnectionStrings:Northwind %>"
    SelectCommand="SELECT CustomerID, ContactName FROM Customers
    OrderBy ContactName" />
    <asp:DropDownList ID="lstCustomers" runat="server"
        DataSourceID="sourceCustomers" DataTextField="ContactName"
        DataValueField="CustomerID" AutoPostBack="True">
    </asp:DropDownList>
```

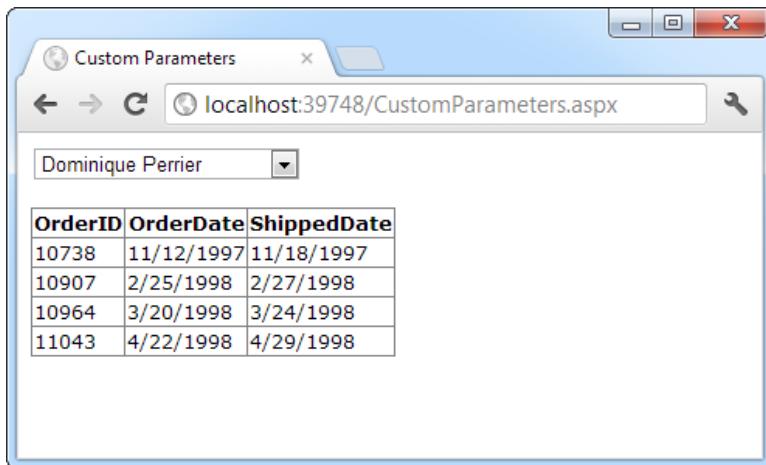


Figure 15-13. Using parameters in a master-details page

When the user picks a customer from the list, the page is posted back (because AutoPostBack is set to true) and the matching orders are shown in a GridView underneath, using a second data source. This data source pulls the CustomerID for the currently selected customer from the drop-down list by using a ControlParameter:

```
<asp:SqlDataSource ID="sourceCustomers" runat="server"
    ProviderName="System.Data.SqlClient"
    ConnectionString="<%$ ConnectionStrings:Northwind %>" 
    SelectCommand="SELECT OrderID,OrderDate,ShippedDate FROM Orders WHERE
CustomerID=@CustomerID">
<SelectParameters>
    <asp:ControlParameter Name="CustomerID"
        ControlID="lstCustomers" PropertyName="SelectedValue" />
</SelectParameters>
</asp:SqlDataSource>

<asp:GridView ID="gridOrders" runat="server" DataSourceID="sourceOrders">
</asp:GridView>
```

Now imagine that you want to limit the order list so it shows only orders made in the last week. This is easy enough to accomplish with a Where clause that examines the OrderDate field. But there's a catch. It doesn't make sense to hard-code the OrderDate value in the query itself, because the range is set based on the current date. And there's no parameter that provides exactly the information you need. The easiest way to solve this problem is to add a new parameter—one that you'll be responsible for setting yourself:

```
<asp:SqlDataSource ID="sourceOrders" runat="server"
    ProviderName="System.Data.SqlClient"
    ConnectionString="<%$ ConnectionStrings:Northwind %>" 
    SelectCommand="SELECT OrderID,OrderDate,ShippedDate FROM Orders WHERE
CustomerID=@CustomerID AND OrderDate >=@EarliestOrderDate"
    OnSelecting="sourceOrders_Selecting">
<SelectParameters>
    <asp:ControlParameter Name="CustomerID"
        ControlID="lstCustomers" PropertyName="SelectedValue" />
```

```
<asp:Parameter Name="EarliestOrderDate" Type="DateTime"
  DefaultValue="1900/01/01" />
</SelectParameters>
</asp:SqlDataSource>
```

Although you can modify the value of any parameter, if you aren't planning to pull the value out of any of the places listed in Table 15-1, it makes sense to use an ordinary Parameter object, as represented by the `<asp:Parameter>` element. You can set the data type (if required) and the default value (as demonstrated in this example).

Now that you've created the parameter, you need to set its value before the command takes place. The `SqlDataSource` has a number of events that are perfect for setting parameter values. You can fill in parameters for a select operation by reacting to the `Selecting` event. Similarly, you can use the `Updating`, `Deleting`, and `Inserting` events when updating, deleting, or inserting a record. In these event handlers, you can access the command that's about to be executed, using the `Command` property of the custom `EventArgs` object (for example, `SqlDataSourceSelectingEventArgs.Command`). You can then modify its parameter values by hand. The `SqlDataSource` also provides similarly named `Selected`, `Updated`, `Deleted`, and `Inserted` events, but these take place after the operation has been completed, so it's too late to change the parameter value.

Here's the code that's needed to set the parameter value to a date that's seven days in the past, ensuring that you see one week's worth of records:

```
protected void sourceOrders_Selecting(object sender,
  SqlDataSourceSelectingEventArgs e)
{
  e.Command.Parameters["@EarliestOrderDate"].Value =
    DateTime.Today.AddDays(-7);
}
```

Note You'll have to tweak this code slightly if you're using it with the standard northwind database. The northwind database is historical, and most orders bear dates around 1997. As a result, the previous code actually retrieve any records. But if you use the `AddYears()` method instead of `AddDays()`, you can easily move 13 years or more, to the place you need to be.

Handling Errors

When you deal with an outside resource such as a database, you need to protect your code with a basic amount of error-handling logic. Even if you've avoided every possible coding mistake, you still need to defend against factors outside your control—for example, if the database server isn't running or the network connection is broken.

You can count on the `SqlDataSource` to properly release any resources (such as connections) if an error occurs. However, the underlying exception won't be handled. Instead, it will bubble up to the page and derail your processing. As with any other unhandled exception, the user will receive a cryptic error message or an error page. This design is unavoidable—if the `SqlDataSource` suppressed exceptions, it could hide potential problems and make debugging extremely difficult. However, it's a good idea to handle the problem in your web page and show a more suitable error message.

To do this, you need to handle the data source event that occurs immediately *after* the error. If you're performing a query, that's the `Selected` event. If you're performing an update, delete, or insert operation, you would handle the `Updated`, `Deleted`, or `Inserted` event instead. (If you don't want to offer customized error messages, you could handle all these events with the same event handler.)

In the event handler, you can access the exception object through the `SqlDataSourceStatusEventArgs.Exception` property. If you want to prevent the error from spreading any further, simply set the `SqlDataSourceStatusEventArgs.ExceptionHandled` property to true. Then make sure you show an appropriate error message on your web page to inform the user that the command was not completed.

Here's an example:

```
protected void sourceProducts_Selected(object sender,
SqlDataSourceStatusEventArgs e)
{
    if (e.Exception != null)
    {
        lblError.Text = "An exception occurred performing the query.";

        // Consider the error handled.
        e.ExceptionHandled = true;
    }
}
```

Updating Records

Selecting data is only half the equation. The SqlDataSource can also apply changes. The only catch is that not all controls support updating. For example, the humble ListBox doesn't provide any way for the user to edit values, delete existing items, or insert new ones. Fortunately, ASP.NET's rich data controls—including the GridView, DetailsView, and FormView—all have editing features you can switch on.

Before you can switch on the editing features in a given control, you need to define suitable commands for the operations you want to perform in your data source. That means supplying commands for inserting (InsertCommand), deleting (DeleteCommand), and updating (UpdateCommand). If you know you will allow the user to perform only certain operations (such as updates) but not others (such as insertions and deletions), you can safely omit the commands you don't need.

You define InsertCommand, DeleteCommand, and UpdateCommand in the same way you define the command for the SelectCommand property—by using a parameterized query. For example, here's a revised version of the SqlDataSource for product information that defines a basic update command to update every field:

```
<asp:SqlDataSource ID="sourceProductDetails" runat="server"
    ProviderName="System.Data.SqlClient"
    ConnectionString="<%$ ConnectionStrings:Northwind %>"
    SelectCommand="SELECT ProductID, ProductName, UnitPrice, UnitsInStock,
    UnitsOnOrder, ReorderLevel, Discontinued FROM Products WHERE ProductID=@ProductID"
    UpdateCommand="UPDATE Products SET ProductName=@ProductName, UnitPrice=@UnitPrice,
    UnitsInStock=@UnitsInStock, UnitsOnOrder=@UnitsOnOrder, ReorderLevel=@ReorderLevel,
    Discontinued=@Discontinued WHERE ProductID=@ProductID">
    <SelectParameters>
        <asp:ControlParameter ControlID="lstProduct" Name="ProductID"
            PropertyName="SelectedValue" />
    </SelectParameters>
</asp:SqlDataSource>
```

In this example, the parameter names aren't chosen arbitrarily. As long as you give each parameter the same name as the field it affects, and preface it with the @ symbol (so ProductName becomes @ProductName), you don't need to define the parameter. That's because the ASP.NET data controls automatically submit a collection of parameters with the new values before triggering the update. Each parameter in the collection uses this naming convention, which is a major time-saver.

You also need to give the user a way to enter the new values. Most rich data controls make this fairly easy—with the DetailsView, it's simply a matter of setting the AutoGenerateEditButton property to true, as shown here:

```
<asp:DetailsView ID="DetailsView1" runat="server"
    DataSourceID="sourceProductDetails" AutoGenerateEditButton="True" />
```

Now when you run the page, you'll see an edit link. When clicked, this link switches the DetailsView into edit mode. All fields are changed to edit controls (typically text boxes), and the Edit link is replaced with an Update link and a Cancel link (see Figure 15-14).

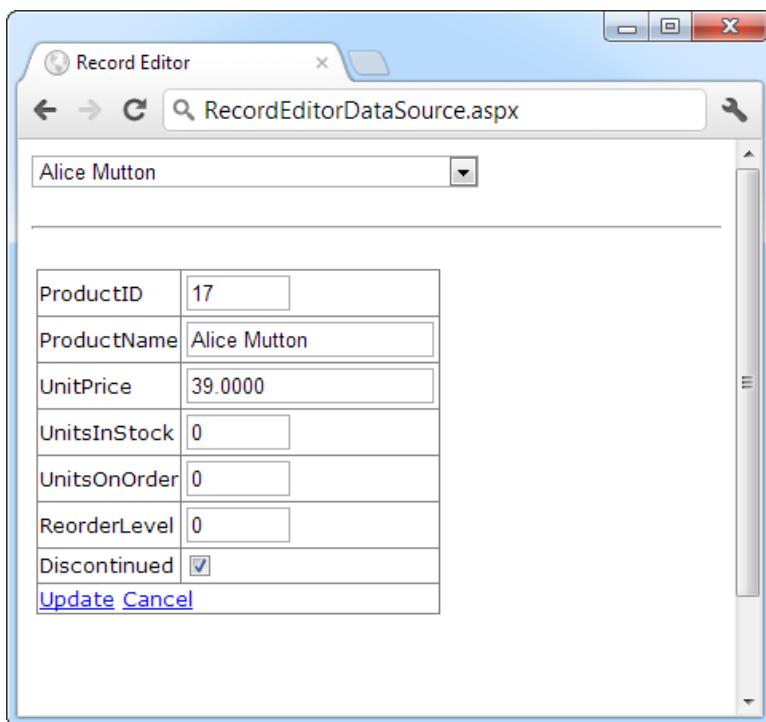


Figure 15-14. Editing with the DetailsView

Clicking the Cancel link returns the row to its initial state. Clicking the Update link triggers an update. The DetailsView extracts the field values, uses them to set the parameters in the `SqlDataSource.UpdateParameters` collection, and then triggers the `SqlDataSource.UpdateCommand` to apply the change to the database. Once again, you don't have to write any code.

You can create similar parameterized commands for `DeleteCommand` and `InsertCommand`. To enable deleting and inserting, you need to set the `AutoGenerateDeleteButton` and `AutoGenerateInsertButton` properties of the DetailsView to true. To see a sample page that allows updating, deleting, and inserting, refer to the `UpdateDeleteInsert.aspx` page that's included with the downloadable samples for this chapter.

Strict Concurrency Checking

The update command in the previous example matches the record based on its ID. You can tell this by examining the `Where` clause:

```
UpdateCommand="UPDATE Products SET ProductName=@ProductName, UnitPrice=@UnitPrice,
    UnitsInStock=@UnitsInStock, UnitsOnOrder=@UnitsOnOrder, ReorderLevel=@ReorderLevel,
    Discontinued=@Discontinued WHERE ProductID=@ProductID"
```

The problem with this approach is that it opens the door to an update that overwrites the changes of another user, if these changes are made between the time your page is requested and the time your page commits its update.

For example, imagine Chen and Lucy are viewing the same table of product records. Lucy commits a change to the price of a product. A few seconds later, Chen commits a name change to the same product record. Chen's update command not only applies the new name but also overwrites all the other fields with the values from Chen's page—replacing the price Lucy entered with the price from the original page.

One way to solve this problem is to use an approach called *match-all-values* concurrency. In this situation, your update command attempts to match every field. As a result, if the original record has changed, the update command won't find it and the update won't be performed at all. So in the scenario described previously, using the match-all-values strategy, Chen would receive an error when he attempts to apply the new product name, and he would need to edit the record and apply the change again.

To use this approach, you need to add a Where clause that tries to match every field. Here's what the modified command would look like:

```
UpdateCommand="UPDATE Products SET ProductName=@ProductName, UnitPrice=@UnitPrice,
UnitsInStock=@UnitsInStock, UnitsOnOrder=@UnitsOnOrder, ReorderLevel=@ReorderLevel,
Discontinued=@Discontinued WHERE ProductID=@ProductID AND
ProductName=@original_ProductName AND UnitPrice=@original_UnitPrice AND
UnitsInStock=@original_UnitsInStock AND UnitsOnOrder=@original_UnitsOnOrder AND
ReorderLevel=@original_ReorderLevel AND Discontinued=@original_Discontinued"
```

Although this makes sense conceptually, you're not finished yet. Before this command can work, you need to tell the SqlDataSource to maintain the old values from the data source and to give them parameter names that start with `original_`. You do this by setting two properties. First, set the `SqlDataSource.ConflictDetection` property to `ConflictOptions.CompareAllValues` instead of `ConflictOptions.OverwriteChanges` (the default). Next, set the long-winded `OldValuesParameterFormatString` property to the text "`original_{0}`". This tells the SqlDataSource to insert the text `original_` before the field name to create the parameter that stores the old value. Now your command will work as written.

The SqlDataSource doesn't raise an exception to notify you if no update is performed. So, if you use the command shown in this example, you need to handle the `SqlDataSourceUpdated` event and check the `SqlDataSourceStatusEventArgs.AffectedRows` property. If it's 0, no records have been updated, and you should notify the user about the concurrency problem so the update can be attempted again, as shown here:

```
protected void sourceProductDetails_Updated(object sender,
    SqlDataSourceStatusEventArgs e)
{
    if (e.AffectedRows == 0)
    {
        lblInfo.Text = "No update was performed. " +
        "A concurrency error is likely, or the command is incorrectly written.";
    }
    else
    {
        lblInfo.Text = "Record successfully updated.";
    }
}
```

Figure 15-15 shows the result you'll get if you run two copies of this page in two separate browser windows, begin editing in both of them, and then try to commit both updates.

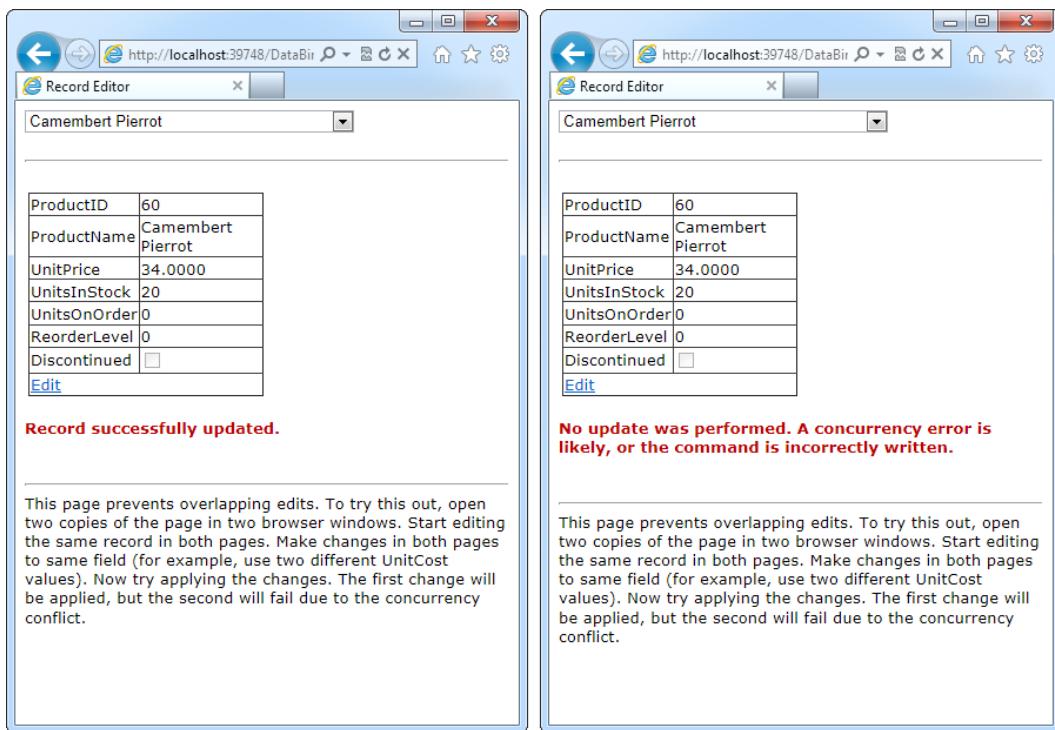


Figure 15-15. A concurrency error in action

Matching every field is an acceptable approach for small records, but it isn't the most efficient strategy if you have tables with huge amounts of data. In this situation, you have two possible solutions: you can match some of the fields (leaving out the ones with really big values) or you can add a timestamp field to your database table, and use that for concurrency checking.

Timestamps are special fields that the database uses to keep track of the state of a record. Whenever any change is made to a record, the database engine updates the timestamp field, giving it a new, automatically generated value. The purpose of a timestamp field is to make strict concurrency checking easier. When you attempt to perform an update to a table that includes a timestamp field, you use a Where clause that matches the appropriate unique ID value (for example, ProductID) and the timestamp field:

```
UpdateCommand="UPDATE Products SET ProductName=@ProductName, UnitPrice=@UnitPrice,
UnitsInStock=@UnitsInStock, UnitsOnOrder=@UnitsOnOrder,
ReorderLevel=@ReorderLevel, Discontinued=@Discontinued
WHERE ProductID=@ProductID AND RowTimestamp=@RowTimestamp"
```

The database engine uses the ProductID to look up the matching record. Then it attempts to match the timestamp in order to update the record. If the timestamp matches, you know the record hasn't been changed. The actual *value* of the timestamp isn't important, because that's controlled by the database. You just need to know whether it has changed.

Creating a timestamp is easy. In SQL Server, you create a timestamp field by using the timestamp data type. In other database products, timestamps are sometimes called *row versions*.

The Last Word

This chapter presented a thorough overview of data binding in ASP.NET. First you learned an interesting way to create dynamic text with simple data binding. Although this is a reasonable approach to get information into your page, it doesn't surpass what you can already do with pure code. You also learned how ASP.NET builds on this infrastructure with much more useful features, including repeated-value binding for quick-and-easy data display in a list control, and data source controls, which let you create code-free bound pages.

Using the techniques in this chapter, you can create a wide range of data-bound pages. However, if you want to create a page that incorporates record editing, sorting, and other more advanced tricks, the data-binding features you've learned about so far are just the first step. You'll also need to turn to specialized controls, such as the DetailsView and the GridView, which build upon these data-binding features. You'll learn how to master these controls in the next chapter. In Chapter 22, you'll learn how to extend your data-binding skills to work with data access components.



The Data Controls

When it comes to data binding, not all ASP.NET controls are created equal. In the previous chapter, you saw how data binding can help you automatically insert single values and lists into all kinds of common controls. In this chapter, you'll concentrate on three more advanced controls—GridView, DetailsView, and FormView—that allow you to bind entire tables of data.

The rich data controls are quite a bit different from the simple list controls. For one thing, they are designed exclusively for data binding. They also have the ability to display more than one field at a time, often in a table-based layout or according to what you've defined. They also support higher-level features such as selecting, editing, and sorting.

The rich data controls include the following:

- *GridView*: The GridView is an all-purpose grid control for showing large tables of information. The GridView is the heavyweight of ASP.NET data controls.
- *DetailsView*: The DetailsView is ideal for showing a single record at a time, in a table that has one row per field. The DetailsView also supports editing.
- *FormView*: Like the DetailsView, the FormView shows a single record at a time and supports editing. The difference is that the FormView is based on templates, which allow you to combine fields in a flexible layout that doesn't need to be table based.
- *ListView*: The ListView plays the same role as the GridView—it allows you to show multiple records. The difference is that the ListView is based on templates. As a result, using the ListView requires a bit more work and gives you slightly more layout flexibility. The ListView isn't described in this book, although you can learn more about it in the Visual Studio Help, or in the book *Pro ASP.NET 4.5 in C#* (Apress).

In this chapter, you'll explore the rich data controls in detail.

The GridView

The GridView is an extremely flexible grid control that displays a multicolumn table. Each record in your data source becomes a separate row in the grid. Each field in the record becomes a separate column in the grid.

The GridView is the most powerful of the rich data controls you'll learn about in this chapter because it comes equipped with the most ready-made functionality. This functionality includes features for automatic paging, sorting, selecting, and editing. The GridView is also the only data control you'll consider in this chapter that can show more than one record at a time.

Automatically Generating Columns

The GridView provides a DataSource property for the data object you want to display, much like the list controls you saw in Chapter 15. Once you've set the DataSource property, you call the DataBind() method to perform the data binding and display each record in the GridView. However, the GridView doesn't provide properties, such as DataTextField and DataValueField, that allow you to choose what column you want to display. That's because the GridView automatically generates a column for *every* field, as long as the AutoGenerateColumns property is true (which is the default).

Here's all you need to create a basic grid with one column for each field:

```
<asp:GridView ID = "GridView1" runat = "server" />
```

Once you've added this GridView tag to your page, you can fill it with data. Here's an example that performs a query using the ADO.NET objects and binds the retrieved DataSet:

```
protected void Page_Load(object sender, EventArgs e)
{
    // Define the ADO.NET objects.
    string connectionString =
        WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
    string selectSQL = "SELECT ProductID, ProductName, UnitPrice FROM Products";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(selectSQL, con);
    SqlDataAdapter adapter = new SqlDataAdapter(cmd);

    // Fill the DataSet.
    DataSet ds = new DataSet();
    adapter.Fill(ds, "Products");

    // Perform the binding.
    GridView1.DataSource = ds;
    GridView1.DataBind();
}
```

Remember, in order for this code to work you must have a connection string named Northwind in the web.config file (just as you did for the examples in the previous two chapters).

Figure 16-1 shows the GridView this code creates.

ProductID	ProductName	UnitPrice
1	Chai	18.0000
2	Chang	19.0000
3	Aniseed Syrup	10.0000
4	Chef Anton's Cajun Seasoning	22.0000
5	Chef Anton's Gumbo Mix	21.3500
6	Grandma's Boysenberry Spread	25.0000
7	Uncle Bob's Organic Dried Pears	30.0000
8	Northwoods Cranberry Sauce	40.0000
9	Mishi Kobe Niku	97.0000
10	Ikura	31.0000
11	Queso Cabrales	21.0000
12	Queso Manchego La Pastora	38.0000
13	Konbu	6.0000
14	Tofu	23.2500
15	Genen Shouyu	15.5000
16	Pavlova	17.4500
17	Alice Mutton	39.0000
18	Carnarvon Tigers	62.5000

Figure 16-1. The bare-bones GridView

Of course, you don't need to write this data access code by hand. As you learned in the previous chapter, you can use the `SqlDataSource` control to define your query. You can then link that query directly to your data control, and ASP.NET will take care of the entire data binding process.

Here's how you would define a `SqlDataSource` to perform the query shown in the previous example:

```
<asp:SqlDataSource ID = "sourceProducts" runat = "server"
ConnectionString = "<%$ ConnectionStrings:Northwind %>" 
SelectCommand = "SELECT ProductID, ProductName, UnitPrice FROM Products" />
```

Next, set the `GridView.DataSourceID` property to link the data source to your grid:

```
<asp:GridView ID = "GridView1" runat = "server"
DataSourceID = "sourceProducts" />
```

These two tags duplicate the example in Figure 16-1 but with significantly less effort. Now you don't have to write any code to execute the query and bind the `DataSet`.

Using the `SqlDataSource` has positive and negative sides. Although it gives you less control, it streamlines your code quite a bit, and it allows you to remove all the database details from your code-behind class. In this chapter, you'll focus on the data source approach because it's much simpler when creating complex data-bound pages that support features such as editing. In Chapter 22, you'll learn how to adapt these examples to use the `ObjectDataSource` instead of the `SqlDataSource`. The `ObjectDataSource` is a great compromise: it allows you to write customized data access code in a database component without giving up the convenient design-time features of the data source controls.

Defining Columns

By default, the `GridView.AutoGenerateColumns` property is true, and the `GridView` creates a column for each field in the bound `DataTable`. This automatic column generation is good for creating quick test pages, but it doesn't give you the flexibility you'll usually want. For example, suppose you want to hide columns, change their order, or configure some aspect of their display, such as the formatting or heading text. In all these cases, you need to set `AutoGenerateColumns` to false and define the columns in the `<Columns>` section of the `GridView` control tag.

Tip It's possible to have `AutoGenerateColumns` set to true and define columns in the `<Columns>` section. In this case, the columns you explicitly define are added before the autogenerated columns. However, for the most flexibility, you'll usually want to explicitly define every column.

Each column can be any of several column types, as described in Table 16-1. The order of your column tags determines the left-to-right order of columns in the `GridView`.

Table 16-1. Column Types

Class	Description
BoundField	This column displays text from a field in the data source.
ButtonField	This column displays a button in this grid column.
CheckBoxField	This column displays a check box in this grid column. It's used automatically for true/false fields (in SQL Server, these are fields that use the bit data type).
CommandField	This column provides selection or editing buttons.
HyperLinkField	This column displays its contents (a field from the data source or static text) as a hyperlink.
ImageField	This column displays image data from a binary field (providing it can be successfully interpreted as a supported image format).
TemplateField	This column allows you to specify multiple fields, custom controls, and arbitrary HTML using a custom template. It gives you the highest degree of control but requires the most work.

The most basic column type is `BoundField`, which binds to one field in the data object. For example, here's the definition for a single data-bound column that displays the `ProductID` field:

```
<asp:BoundField DataField = "ProductID" HeaderText = "ID" />
```

This tag demonstrates how you can change the header text at the top of a column from `ProductID` to just `ID`. Here's a complete `GridView` declaration with explicit columns:

```
<asp:GridView ID = "GridView1" runat = "server" DataSourceID = "sourceProducts"
AutoGenerateColumns = "False">
<Columns>
```

```
<asp:BoundField DataField = "ProductID" HeaderText = "ID" />
<asp:BoundField DataField = "ProductName" HeaderText = "Product Name" />
<asp:BoundField DataField = "UnitPrice" HeaderText = "Price" />
</Columns>
</asp:GridView>
```

Explicitly defining the columns has several advantages:

- You can easily fine-tune your column order, column headings, and other details by tweaking the properties of your column object.
- You can hide columns you don't want to show by removing the column tag. (Don't overuse this technique, because it's better to reduce the amount of data you're retrieving if you don't intend to display it.)
- You'll see your columns in the design environment (in Visual Studio). With automatically generated columns, the GridView simply shows a few generic placeholder columns.
- You can add extra columns to the mix for selecting, editing, and more.

This example shows how you can use this approach to change the header text. However, the HeaderText property isn't the only column property you can change in a column. In the next section, you'll learn about a few more.

Configuring Columns

When you explicitly declare a bound field, you have the opportunity to set other properties. Table 16-2 lists these properties.

Table 16-2. *BoundField Properties*

Property	Description
DataField	Identifies the field (by name) that you want to display in this column.
DataFormatString	Formats the field. This is useful for getting the right representation of numbers and dates.
ApplyFormatInEditMode	If true, the DataFormat string is used to format the value even when the value appears in a text box in edit mode. The default is false, which means the underlying value will be used (such as 1143.02 instead of \$1,143.02).
FooterText, HeaderText, and HeaderImageUrl	Sets the text in the header and footer region of the grid if this grid has a header (GridView.ShowHeader is true) and footer (GridView.ShowFooter is true). The header is most commonly used for a descriptive label such as the field name; the footer can contain a dynamically calculated value such as a summary. To show an image in the header <i>instead</i> of text, set the HeaderImageUrl property.
ReadOnly	If true, it prevents the value for this column from being changed in edit mode. No edit control will be provided. Primary key fields are often read-only.
InsertVisible	If true, it prevents the value for this column from being set in insert mode. If you want a column value to be set programmatically or based on a default value defined in the database, you can use this feature.

(continued)

Table 16-2. (continued)

Property	Description
Visible	If false, the column won't be visible in the page (and no HTML will be rendered for it). This gives you a convenient way to programmatically hide or show specific columns, changing the overall view of the data.
SortExpression	Sorts your results based on one or more columns. You'll learn about sorting later in the "Sorting and Paging the GridView" section of this chapter.
HtmlEncode	If true (the default), all text will be HTML encoded to prevent special characters from mangling the page. You could disable HTML encoding if you want to embed a working HTML tag (such as a hyperlink), but this approach isn't safe. It's always better to use HTML encoding on all values and provide other functionality by reacting to GridView selection events.
NullDisplayText	Displays the text that will be shown for a null value. The default is an empty string, although you could change this to a hard-coded value, such as "(not specified)."
ConvertEmptyStringToNull	If true, converts all empty strings to null values (and uses the NullDisplayText to display them).
ControlStyle, HeaderStyle, FooterStyle, and ItemStyle	Configures the appearance for just this column, overriding the styles for the row. You'll learn more about styles throughout this chapter.

Generating Columns with Visual Studio

As you've already learned, you can create a GridView that shows all your fields by setting the AutoGenerateColumns property to true. Unfortunately, when you use this approach you lose the ability to control any of the details regarding your columns, including their order, formatting, sorting, and so on. To configure these details, you need to set AutoGenerateColumns to false and define your columns explicitly. This requires more work, and it's a bit tedious.

However, there is a nifty trick that solves this problem. You can use explicit columns but get Visual Studio to create the column tags for you automatically. Here's how it works: select the GridView control, and click Refresh Schema in the smart tag. At this point, Visual Studio will retrieve the basic schema information from your data source (for example, the names and data type of each column) and then add one < BoundField > element for each field.

Tip If you modify the data source so it returns a different set of columns, you can regenerate the Grid columns. Just select the GridView, and click the Refresh schema link in the smart tag. This step will wipe out custom columns you've added (such as editing controls).

Once you've created your columns, you can also use some helpful design-time support to configure the properties of each column (rather than editing the column tag by hand). To do this, select the GridView and click the ellipsis (. . .) next to the Columns property in the Properties window. You'll see a Fields dialog box that lets you add, remove, and refine your columns (see Figure 16-2).

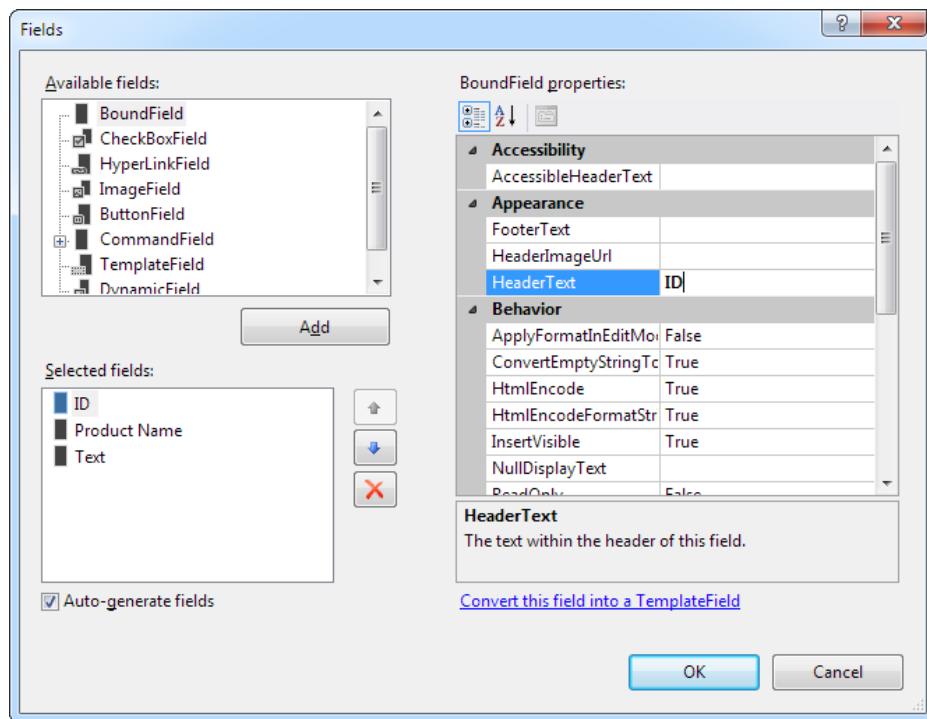


Figure 16-2. Configuring columns in Visual Studio

Now that you understand the underpinnings of the GridView, you've begun to explore some of its higher-level features. In the following sections, you'll tackle these topics:

Formatting: How to format rows and data values

Selecting: How to let users select a row in the GridView and respond accordingly

Editing: How to let users commit record updates, inserts, and deletes

Sorting: How to dynamically reorder the GridView in response to clicks on a column header

Paging: How to divide a large result set into multiple pages of data

Templates: How to take complete control of designing, formatting, and editing by defining templates

Formatting the GridView

Formatting consists of several related tasks. First, you want to ensure that dates, currencies, and other number values are presented in the appropriate way. You handle this job with the `DataFormatString` property. Next, you'll want to apply the perfect mix of colors, fonts, borders, and alignment options to each aspect of the grid, from headers to data items. The `GridView` supports these features through styles. Finally, you can intercept events, examine row data, and apply formatting to specific values programmatically. In the following sections, you'll consider each of these techniques.

The `GridView` also exposes several self-explanatory formatting properties that aren't covered here. These include `GridLines` (for adding or hiding table borders), `CellPadding` and `CellSpacing` (for controlling the overall spacing between cells), and `Caption` and `CaptionAlign` (for adding a title to the top of the grid).

Tip Want to create a `GridView` that scrolls—inside a web page? It's easy. Just place the `GridView` inside a `Panel` control, set the appropriate size for the panel, and set the `Panel.ScrollBars` property to `Auto`, `Vertical`, or `Both`.

Formatting Fields

Each `BoundField` column provides a `DataFormatString` property you can use to configure the appearance of numbers and dates using a *format string*.

Format strings generally consist of a placeholder and a format indicator, which are wrapped inside curly brackets. A typical format string looks something like this:

```
{0:C}
```

In this case, the 0 represents the value that will be formatted, and the letter indicates a predetermined format style. Here, C means currency format, which formats a number as an amount of money (so 3400.34 becomes \$3,400.34, assuming the web server is set to use U.S. regional settings). Here's a column that uses this format string:

```
<asp:BoundField DataField = "UnitPrice" HeaderText = "Price"
DataFormatString = "{0:C}" />
```

Table 16-3 shows some of the other formatting options for numeric values.

Table 16-3. Numeric Format Strings

Type	Format String	Example
Currency	{0:C}	\$1,234.50. Brackets indicate negative values: (\$1,234.50). The currency sign is locale-specific.
Scientific (Exponential)	{0:E}	1.234500E + 003
Percentage	{0:P}	45.6 %
Fixed Decimal	{0:F?}	Depends on the number of decimal places you set. {0:F3} would be 123.400. {0:F0} would be 123.

You can find other examples in the MSDN Help. For date or time values, you'll find an extensive list. For example, if you want to write the BirthDate value in the format month/day/year (as in 12/30/12), you use the following column:

```
<asp:BoundField DataField = "BirthDate" HeaderText = "Birth Date"
DataFormatString = "{0:MM/dd/yy}" />
```

Table 16-4 shows some more examples.

Table 16-4. Time and Date Format Strings

Type	Format String	Syntax	Example
Short Date	{0:d}	M/d/yyyy	10/30/2012
Long Date	{0:D}	dddd, MMMM dd, yyyy	Monday, January 30, 2012
Long Date and Short Time	{0:f }	dddd, MMMM dd, yyyy HH:mm aa	Monday, January 30, 2012 10:00 AM
Long Date and Long Time	{0:F}	dddd, MMMM dd, yyyy HH:mm:ss aa	Monday, January 30, 2012 10:00:23 AM
ISO Sortable Standard	{0:s}	yyyy-MM-ddTHH:mm:ss	2012-01-30 T10:00:23
Month and Day	{0:M}	MMMM dd	January 30
General	{0:G}	M/d/yyyy HH:mm:ss aa (depends on locale-specific settings)	10/30/2012 10:00:23 AM

The format characters are not specific to the GridView. You can use them with other controls, with data-bound expressions in templates (as you'll see later in the "Using GridView Templates" section), and as parameters for many methods. For example, the Decimal and DateTime types expose their own `ToString()` methods that accept a format string, allowing you to format values manually.

Using Styles

The GridView exposes a rich formatting model that's based on *styles*. Altogether, you can set eight GridView styles, as described in Table 16-5.

Table 16-5. GridView Styles

Style	Description
HeaderStyle	Configures the appearance of the header row that contains column titles, if you've chosen to show it (if ShowHeader is true).
RowStyle	Configures the appearance of every data row.
AlternatingRowStyle	If set, applies additional formatting to every other row. This formatting acts in addition to the RowStyle formatting. For example, if you set a font using RowStyle, it is also applied to alternating rows, unless you explicitly set a different font through AlternatingRowStyle.
SelectedRowStyle	Configures the appearance of the row that's currently selected. This formatting acts in addition to the RowStyle formatting.
EditRowStyle	Configures the appearance of the row that's in edit mode. This formatting acts in addition to the RowStyle formatting.
EmptyDataRowStyle	Configures the style that's used for the single empty row in the special case where the bound data object contains no rows.
FooterStyle	Configures the appearance of the footer row at the bottom of the GridView, if you've chosen to show it (if ShowFooter is true).
PagerStyle	Configures the appearance of the row with the page links, if you've enabled paging (set AllowPaging to true).

Styles are not simple single-value properties. Instead, each style exposes a Style object that includes properties for choosing colors (ForeColor and BackColor), adding borders (BorderColor, BorderStyle, and BorderWidth), sizing the row (Height and Width), aligning the row (HorizontalAlign and VerticalAlign), and configuring the appearance of text (Font and Wrap). These style properties allow you to refine almost every aspect of an item's appearance.

Here's an example that changes the style of rows and headers in a GridView:

```
<asp:GridView ID = "GridView1" runat = "server" DataSourceID = "sourceProducts"
AutoGenerateColumns = "False">
<RowStyle BackColor = "#E7E7FF" ForeColor = "#4A3C8C" />
<HeaderStyle BackColor = "#4A3C8C" Font-Bold = "True" ForeColor = "#F7F7F7" />
<Columns>
<asp:BoundField DataField = "ProductID" HeaderText = "ID" />
<asp:BoundField DataField = "ProductName" HeaderText = "Product Name" />
<asp:BoundField DataField = "UnitPrice" HeaderText = "Price" />
</Columns>
</asp:GridView>
```

In this example, every column is affected by the formatting changes. However, you can also define column-specific styles. To create a column-specific style, you simply need to rearrange the control tag so that the formatting tag becomes a nested tag *inside* the appropriate column tag. Here's an example that formats just the ProductName column:

```
<asp:GridView ID = "GridView2" runat = "server" DataSourceID = "sourceProducts"
AutoGenerateColumns = "False" >
<Columns>
<asp:BoundField DataField = "ProductID" HeaderText = "ID" />
<asp:BoundField DataField = "ProductName" HeaderText = "Product Name">
    <ItemStyle BackColor = "#E7E7FF" ForeColor = "#4A3C8C" />
    <HeaderStyle BackColor = "#4A3C8C" Font-Bold = "True" ForeColor = "#F7F7F7" />
</asp:BoundField>
<asp:BoundField DataField = "UnitPrice" HeaderText = "Price" />
</Columns>
</asp:GridView>
```

Figure 16-3 compares these two examples. You can use a combination of ordinary style settings and column-specific style settings (which override ordinary style settings if they conflict).

Global style settings:		
ID	Product Name	Price
1	Chai	18.0000
2	Chang	19.0000
3	Aniseed Syrup	10.0000
4	Chef Anton's Cajun Seasoning	22.0000
5	Chef Anton's Gumbo Mix	21.3500
6	Grandma's Boysenberry Spread	25.0000
7	Uncle Bob's Organic Dried Pears	30.0000
8	Northwoods Cranberry Sauce	40.0000
9	Mishi Kobe Niku	97.0000
10	Ikura	31.0000
11	Queso Cabrales	21.0000
12	Queso Manchego La Pastora	38.0000
13	Konbu	6.0000
14	Tofu	23.2500
15	Genen Shouyu	15.5000
16	Pavlova	17.4500
17	Alice Mutton	39.0000
18	Carnarvon Tigers	62.5000
19	Teatime Chocolate Biscuits	9.2000
20	Sir Rodney's Marmalade	81.0000
21	Sir Rodney's Scones	10.0000

Column-specific styles:		
ID	Product Name	Price
1	Chai	18.0000
2	Chang	19.0000
3	Aniseed Syrup	10.0000
4	Chef Anton's Cajun Seasoning	22.0000
5	Chef Anton's Gumbo Mix	21.3500
6	Grandma's Boysenberry Spread	25.0000
7	Uncle Bob's Organic Dried Pears	30.0000
8	Northwoods Cranberry Sauce	40.0000
9	Mishi Kobe Niku	97.0000
10	Ikura	31.0000
11	Queso Cabrales	21.0000
12	Queso Manchego La Pastora	38.0000
13	Konbu	6.0000
14	Tofu	23.2500
15	Genen Shouyu	15.5000
16	Pavlova	17.4500
17	Alice Mutton	39.0000
18	Carnarvon Tigers	62.5000
19	Teatime Chocolate Biscuits	9.2000
20	Sir Rodney's Marmalade	81.0000
21	Sir Rodney's Scones	10.0000

Figure 16-3. Formatting the GridView

One reason you might use column-specific formatting is to define specific column widths. If you don't define a specific column width, ASP.NET makes each column just wide enough to fit the data it contains (or, if wrapping is enabled, to fit the text without splitting a word over a line break). If values range in size, the width is determined by the largest value or the width of the column header, whichever is larger. However, if the grid is wide enough, you might want to expand a column so it doesn't appear to be crowded against the adjacent columns. In this case, you need to explicitly define a larger width.

Configuring Styles with Visual Studio

There's no reason to code style properties by hand in the GridView control tag because the GridView provides rich design-time support. To set style properties, you can use the Properties window to modify the style properties. For example, to configure the font of the header, expand the HeaderStyle property to show the nested Font property, and set that. The only limitation on this approach is that it doesn't allow you to set the style for individual columns; if you need that trick, you must first call up the Fields dialog box (shown in Figure 16-2) by editing the Columns property. Then, select the appropriate column and set the style properties accordingly.

You can even set a combination of styles using a preset theme by clicking the Auto Format link in the GridView smart tag. Figure 16-4 shows the Auto Format dialog box with some of the preset styles you can choose. Select Remove Formatting to clear all the style settings.

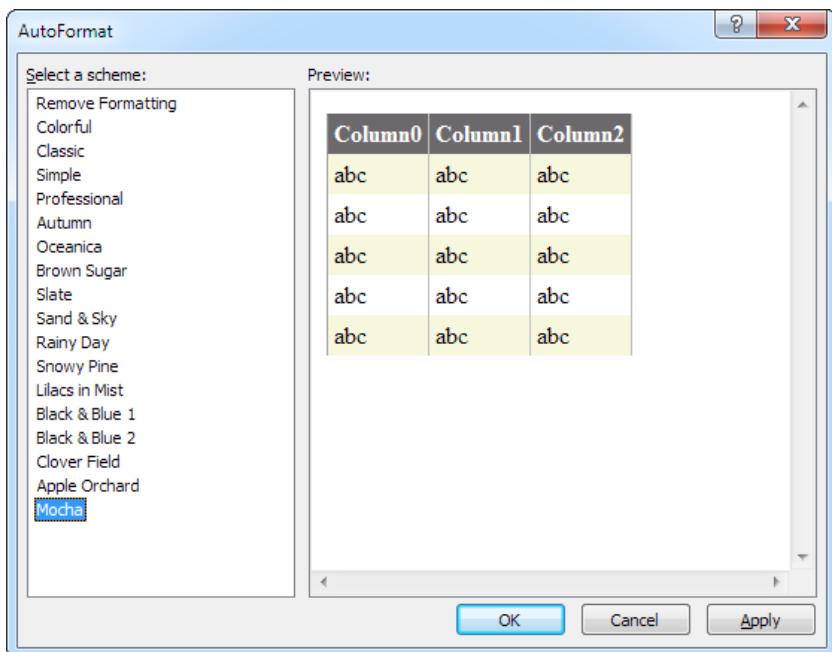


Figure 16-4. Automatically formatting a GridView

Once you've chosen and inserted the styles into your `GridView` tag, you can tweak them by hand or by using the Properties window.

Formatting-Specific Values

The formatting you've learned so far isn't that fine-grained. At its most specific, this formatting applies to a single column of values. But what if you want to change the formatting for a specific row or even for just a single cell?

The solution is to react to the `GridView.RowDataBound` event. This event is raised for each row, just after it's filled with data. At this point, you can access the current row as a `GridViewRow` object. The `GridViewRow.DataItem` property provides the data object for the given row, and the `GridViewRow.Cells` collection allows you to retrieve the row content. You can use the `GridViewRow` to change colors and alignment, add or remove child controls, and so on.

The following example handles the `RowDataBound` event and changes the background color to highlight high prices (those more expensive than \$50):

```
protected void GridView1_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // Get the price for this row.
        decimal price = (decimal)DataBinder.Eval(e.Row.DataItem, "UnitPrice");

        if (price > 50)
        {
            e.Row.BackColor = System.Drawing.Color.Maroon;
            e.Row.ForeColor = System.Drawing.Color.White;
            e.Row.Font.Bold = true;
        }
    }
}
```

First, the code checks whether the item being created is a row or an alternate row. If neither, it means the item is another interface element, such as the pager, footer, or header, and the procedure does nothing. If the item is the right type, the code extracts the `UnitPrice` field from the data-bound item.

To get a value from the bound data object (provided through the `GridViewRowEventArgs.Row.DataItem` property), you need to cast the data object to the correct type. The trick is that the type depends on the way you're performing your data binding. In this example, you're binding to the `SqlDataSource` in `DataSet` mode, which means each data item will be a `DataRowView` object. (If you were to bind in `DataReader` mode, a `DbDataRecord` represents each item instead.) To avoid coding these details, which can make it more difficult to change your data access code, you can rely on the `DataBinder.Eval()` helper method, which understands all these types of data objects. That's the technique used in this example.

Figure 16-5 shows the resulting page.

ID	Product Name	Price
1	Chai	\$18.00
2	Chang	\$19.00
3	Aniseed Syrup	\$10.00
4	Chef Anton's Cajun Seasoning	\$22.00
5	Chef Anton's Gumbo Mix	\$21.35
6	Grandma's Boysenberry Spread	\$25.00
7	Uncle Bob's Organic Dried Pears	\$30.00
8	Northwoods Cranberry Sauce	\$40.00
9	Mishi Kobe Niku	\$97.00
10	Ikura	\$31.00
11	Queso Cabrales	\$21.00
12	Queso Manchego La Pastora	\$38.00
13	Konbu	\$6.00
14	Tofu	\$23.25
15	Genen Shouyu	\$15.50
16	Pavlova	\$17.45
17	Alice Mutton	\$39.00
18	Carnarvon Tigers	\$62.50
19	Teatime Chocolate Biscuits	\$9.20
20	Sir Rodney's Marmalade	\$81.00
21	Sir Rodney's Scones	\$10.00
22	Gustaf's Knäckebröd	\$21.00
23	Tunnbröd	\$9.00
24	Guaraná Fantástica	\$4.50
25	NuNuCa Nuß-Nougat-Creme	\$14.00

Figure 16-5. Formatting individual rows based on values

Selecting a GridView Row

Selecting an item refers to the ability to click a row and have it change color (or become highlighted) to indicate that the user is currently working with this record. At the same time, you might want to display additional information about the record in another control. With the GridView, selection happens almost automatically once you set up a few basics.

Before you can use item selection, you must define a different style for selected items. The `SelectedRowStyle` determines how the selected row or cell will appear. If you don't set this style, it will default to the same value as `RowStyle`, which means the user won't be able to tell which row is currently selected. Usually, selected rows will have a different `BackColor` property.

To find out what item is currently selected (or to change the selection), you can use the `GridView`. `SelectedIndex` property. It will be -1 if no item is currently selected. Also, you can react to the `SelectedIndexChanged` event to handle any additional related tasks. For example, you might want to update another control with additional information about the selected record.

Adding a Select Button

The GridView provides built-in support for selection. You simply need to add a CommandField column with the ShowSelectButton property set to true. ASP.NET can render the CommandField as a hyperlink, a button, or a fixed image. You choose the type using the ButtonType property. You can then specify the text through the SelectText property or specify the link to the image through the SelectImageUrl property.

Here's an example that displays a select button:

```
<asp:CommandField ShowSelectButton = "True" ButtonType = "Button"
SelectText = "Select" />
```

And here's an example that shows a small clickable icon:

```
<asp:CommandField ShowSelectButton = "True" ButtonType = "Image"
SelectImageUrl = "select.gif" />
```

Figure 16-6 shows a page with a text select button (and product 14 selected).

	ProductID	ProductName	UnitPrice
Select	1	Chai	18.0000
Select	2	Chang	19.0000
Select	3	Aniseed Syrup	10.0000
Select	4	Chef Anton's Cajun Seasoning	22.0000
Select	5	Chef Anton's Gumbo Mix	21.3500
Select	6	Grandma's Boysenberry Spread	25.0000
Select	7	Uncle Bob's Organic Dried Pears	30.0000
Select	8	Northwoods Cranberry Sauce	40.0000
Select	9	Mishi Kobe Niku	97.0000
Select	10	Ikura	31.0000
Select	11	Queso Cabrales	21.0000
Select	12	Queso Manchego La Pastora	38.0000
Select	13	Konbu	6.0000
Select	14	Tofu	23.2500
Select	15	Genen Shouyu	15.5000
Select	16	Pavlova	17.4500
Select	17	Alice Mutton	39.0000
Select	18	Carnarvon Tigers	62.5000
Select	19	Teatime Chocolate Biscuits	9.2000

Figure 16-6. GridView selection

When you click a select button, the page is posted back and a series of steps unfolds. First, the GridView.SelectedIndexChanged event fires, which you can intercept to cancel the operation. Next, the GridView.SelectedIndex property is adjusted to point to the selected row. Finally, the GridView.SelectedIndexChanged event fires, which you can handle if you want to manually update other controls to reflect the new selection. When the page is rendered, the selected row is given the selected row style.

Tip Rather than add the select button yourself, you can choose Enable selection from the GridView's `Selecting` which adds a basic select button for you.

Using a Data Field as a Select Button

You don't need to create a new column to support row selection. Instead, you can turn an existing column into a link. This technique is commonly implemented to allow users to select rows in a table by the unique ID value.

To use this technique, remove the CommandField column and add a ButtonField column instead. Then, set the DataTextField to the name of the field you want to use.

```
<asp:ButtonField ButtonType = "Button" DataTextField = "ProductID" />
```

This field will be underlined and turned into a button that, when clicked, will post back the page and trigger the GridView.RowCommand event. You could handle this event, determine which row has been clicked, and programmatically set the SelectedIndex property of the GridView. However, you can use an easier method. Instead, just configure the link to raise the SelectedIndexChanged event by specifying a CommandName with the text *Select*, as shown here:

```
<asp:ButtonField CommandName = "Select" ButtonType = "Button"
DataTextField = "ProductID" />
```

Now, clicking the data field automatically selects the record.

Using Selection to Create Master-Details Pages

As demonstrated in the previous chapter, you can draw a value out of a control and use it to perform a query in your data source. For example, you can take the currently selected item in a list, and feed that value to a SqlDataSource that gets more information for the corresponding record.

This trick is a great way to build *master-details pages*—pages that let you navigate relationships in a database. A typical master-details page has two GridView controls. The first GridView shows the master (or parent) table. When a user selects an item in the first GridView, the second GridView is filled with related records from the details (or parent) table. For example, a typical implementation of this technique might have a Customers table in the first GridView. Select a customer, and the second GridView is filled with the list of orders made by that customer.

To create a master-details page, you need to extract the SelectedIndex property from the first GridView and use that to craft a query for the second GridView. However, this approach has one problem. SelectedIndex returns a zero-based index number that represents where the row occurs in the grid. This isn't the information you need to insert into the query that gets the related records. Instead, you need a unique key field from the corresponding row. For example, if you have a table of products, you need to be able to get the ProductID for the selected row. In order to get this information, you need to tell the GridView to keep track of the key field values.

The way you do this is by setting the DataKeyNames property for the GridView. This property requires a comma-separated list of one or more key fields. Each name you supply must match one of the fields in the bound

data source. Usually, you'll have only one key field. Here's an example that tells the GridView to keep track of the CategoryID values in a list of product categories:

```
<asp:GridView ID = "gridCategories" runat = "server"
DataKeyNames = "CategoryID" . . . >
```

Once you've established this link, the GridView is nice enough to keep track of the key fields for the selected record. It allows you to retrieve this information at any time through the SelectedDataKey property.

The following example puts it all together. It defines two GridView controls. The first shows a list of categories. The second shows the products that fall into the currently selected category (or, if no category has been selected, this GridView doesn't appear at all).

Here's the page markup for this example:

```
Categories:<br />
<asp:GridView ID = "gridCategories" runat = "server" DataSourceID = "sourceCategories"
DataKeyNames = "CategoryID">
<Columns>
<asp:CommandField ShowSelectButton = "True" />
</Columns>
<SelectedRowStyle BackColor = "#FFCC66" Font-Bold = "True"
ForeColor = "#663399" />
</asp:GridView>
<asp:SqlDataSource ID = "sourceCategories" runat = "server"
ConnectionString = "<%$ ConnectionStrings:Northwind %>" 
SelectCommand = "SELECT * FROM Categories" ></asp:SqlDataSource>
<br />

Products in this category:<br />
<asp:GridView ID = "gridProducts" runat = "server" DataSourceID = "sourceProducts">
<SelectedRowStyle BackColor = "#FFCC66" Font-Bold = "True" ForeColor = "#663399" />
</asp:GridView>
<asp:SqlDataSource ID = "sourceProducts" runat = "server"
ConnectionString = "<%$ ConnectionStrings:Northwind %>" 
SelectCommand = "SELECT ProductID, ProductName, UnitPrice FROM Products WHERE
CategoryID = @CategoryID">
<SelectParameters>
<asp:ControlParameter Name = "CategoryID" ControlID = "gridCategories"
PropertyName = "SelectedDataKey.Value" />
</SelectParameters>
</asp:SqlDataSource>
```

As you can see, you need two data sources, one for each GridView. The second data source uses a ControlParameter that links it to the SelectedDataKey property of the first GridView. Best of all, you still don't need to write any code or handle the SelectedIndexChanged event on your own.

Figure 16-7 shows this example in action.

	CategoryID	CategoryName	Description
Select	1	Beverages	Soft drinks, coffees, teas, beers, and ales
Select	2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
Select	3	Confections	Desserts, candies, and sweet breads
Select	4	Dairy Products	Cheeses
Select	5	Grains/Cereals	Breads, crackers, pasta, and cereal
Select	6	Meat/Poultry	Prepared meats
Select	7	Produce	Dried fruit and bean curd
Select	8	Seafood	Seaweed and fish

ProductID	ProductName	UnitPrice
9	Mishi Kobe Niku	97.0000
17	Alice Mutton	39.0000
29	Thüringer Rostbratwurst	123.7900
53	Perth Pasties	32.8000
54	Tourtière	7.4500
55	Pâté chinois	24.0000

Figure 16-7. A master-details page

Editing with the GridView

The GridView provides support for editing that's almost as convenient as its support for selection. To switch a row into select mode, you simply set the SelectedIndex property to the corresponding row number. To switch a row into edit mode, you set the EditIndex property in the same way.

Of course, both of these tasks can take place automatically if you use specialized button types. For selection, you use a CommandField column with the ShowSelectButton property set to true. To add edit controls, you follow almost the same step—once again, you use the CommandField column, but now you set ShowEditButton to true.

Here's an example of a GridView that supports editing:

```
<asp:GridView ID = "gridProducts" runat = "server" DataSourceID = "sourceProducts"
AutoGenerateColumns = "False" DataKeyNames = "ProductID">
<Columns>
<asp:BoundField DataField = "ProductID" HeaderText = "ID" ReadOnly = "True" />
<asp:BoundField DataField = "ProductName" HeaderText = "Product Name"/>
<asp:BoundField DataField = "UnitPrice" HeaderText = "Price" />
<asp:CommandField ShowEditButton = "True" />
</Columns>
</asp:GridView>
```

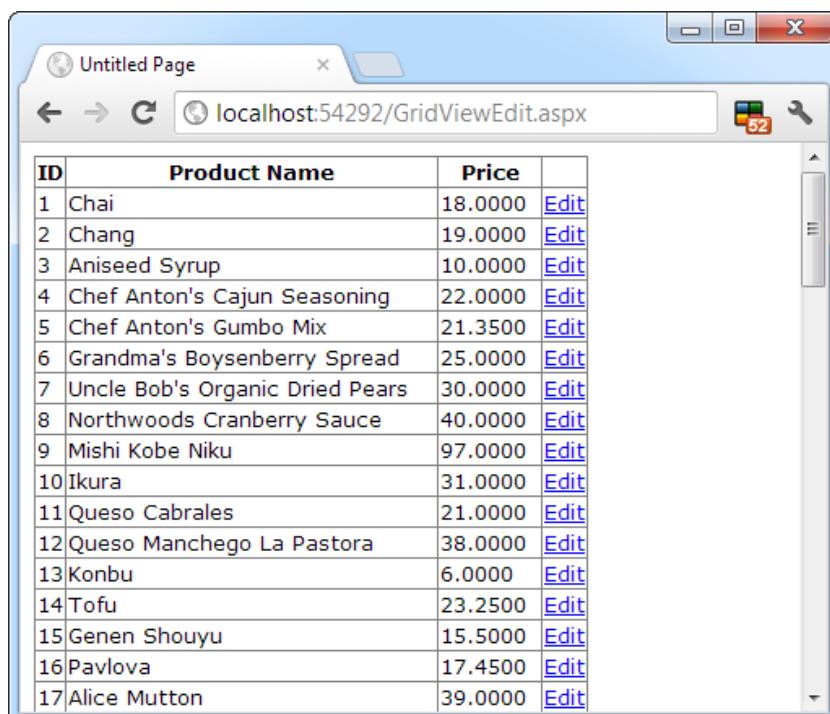
And here's a revised data source control that can commit your changes:

```
<asp:SqlDataSource id = "sourceProducts" runat = "server"
    ConnectionString = "<%$ ConnectionStrings:Northwind %>" 
    SelectCommand = "SELECT ProductID, ProductName, UnitPrice FROM Products"
    UpdateCommand = "UPDATE Products SET ProductName = @ProductName,
    UnitPrice = @UnitPrice WHERE ProductID = @ProductID" />
```

Note If you receive a `sqlException` that says "Must declare the scalar variable @ProductID," the most likely problem is that you haven't set the `GridView.DataKeynames` property. Because the `ProductID` field can't be used by the `GridView` won't pass the `ProductID` value to the `SqlDataSource` unless it's designated a key field.

Remember, you don't need to define the update parameters as long as you make sure they match the field names (with an *at* sign [`@`] at the beginning). Chapter 15 has more information about using update commands with the `SqlDataSource` control.

When you add a `CommandField` with the `ShowEditButton` property set to true, the `GridView` editing controls appear in an additional column. When you run the page and the `GridView` is bound and displayed, the edit column shows an `Edit` link next to every record (see Figure 16-8).



The screenshot shows a Microsoft Internet Explorer window titled "Untitled Page". The address bar displays "localhost:54292/GridViewEdit.aspx". The main content area contains a `GridView` control. The grid has four columns: "ID", "Product Name", "Price", and an empty fourth column. Each row contains a data item from the Northwind database. In the fourth column of each row, there is a blue "Edit" link underlined in blue, indicating an edit control for that specific row. The grid is scrollable, as indicated by the vertical scrollbar on the right side.

ID	Product Name	Price	
1	Chai	18.0000	Edit
2	Chang	19.0000	Edit
3	Aniseed Syrup	10.0000	Edit
4	Chef Anton's Cajun Seasoning	22.0000	Edit
5	Chef Anton's Gumbo Mix	21.3500	Edit
6	Grandma's Boysenberry Spread	25.0000	Edit
7	Uncle Bob's Organic Dried Pears	30.0000	Edit
8	Northwoods Cranberry Sauce	40.0000	Edit
9	Mishi Kobe Niku	97.0000	Edit
10	Ikura	31.0000	Edit
11	Queso Cabrales	21.0000	Edit
12	Queso Manchego La Pastora	38.0000	Edit
13	Konbu	6.0000	Edit
14	Tofu	23.2500	Edit
15	Genen Shouyu	15.5000	Edit
16	Pavlova	17.4500	Edit
17	Alice Mutton	39.0000	Edit

Figure 16-8. The editing controls

When clicked, this link switches the corresponding row into edit mode. All fields are changed to text boxes, with the exception of read-only fields (which are not editable) and true/false bit fields (which are shown as check boxes). The Edit link is replaced with an Update link and a Cancel link (see Figure 16-9).

ID	Product Name	Price	
1	Chai	18.0000	Edit
2	Chang	19.0000	Edit
3	Aniseed Syrup	10.0000	Update Cancel
4	Chef Anton's Cajun Seasoning	22.0000	Edit
5	Chef Anton's Gumbo Mix	21.3500	Edit
6	Grandma's Boysenberry Spread	25.0000	Edit
7	Uncle Bob's Organic Dried Pears	30.0000	Edit
8	Northwoods Cranberry Sauce	40.0000	Edit
9	Mishi Kobe Niku	97.0000	Edit
10	Ikura	31.0000	Edit
11	Queso Cabrales	21.0000	Edit
12	Queso Manchego La Pastora	38.0000	Edit
13	Konbu	6.0000	Edit
14	Tofu	23.2500	Edit
15	Genen Shouyu	15.5000	Edit
16	Pavlova	17.4500	Edit

Figure 16-9. Editing a record

The Cancel link returns the row to its initial state. The Update link passes the values to the `SqlDataSource`.`UpdateParameters` collection (using the field names) and then triggers the `SqlDataSource.Update()` method to apply the change to the database. Once again, you don't have to write any code, provided you've filled in the `UpdateCommand` for the linked data source control.

You can use a similar approach to add support for record deleting. To enable deleting, you need to add a column to the `GridView` that has the `ShowDeleteButton` property set to true. As long as your linked `SqlDataSource` has the `DeleteCommand` property filled in, these operations will work automatically. If you want to write your own code that plugs into this process (for example, updating a label to inform the user the update has been made), consider reacting to the `GridView` event that fires after an update operation is committed, such as `RowDeleted` and `RowUpdated`. You can also prevent changes you don't like by reacting to the `RowDeleting` and `RowUpdating` events and setting the cancel flag in the event arguments.

The `GridView` does not support inserting records. If you want that ability, you can use one of the single-record display controls described later in this chapter, such as the `DetailsView` or `FormView`. For example, a typical ASP.NET page for data entry might show a list of records in a `GridView` and provide a `DetailsView` that allows the user to add new records.

Note The basic built-in updating features of the GridView don't give you a lot of flexibility. You can't change the types of controls that are used for editing, format these controls, or add validation. However, you can add more features by building your own editing templates, a topic presented later in the "Using GridView Templates" section.

Sorting and Paging the GridView

The GridView is a great all-in-one solution for displaying all kinds of data, but it becomes a little unwieldy as the number of fields and rows in your data source grows. Dense grids contribute to large pages that are slow to transmit over the network and difficult for the user to navigate. The GridView has two features that address these issues and make data more manageable: sorting and paging.

Both sorting and paging can be performed by the database server, provided you craft the right SQL using the Order By and Where clauses. In fact, sometimes this is the best approach for performance. However, the sorting and paging provided by the GridView and SqlDataSource are easy to implement and thoroughly flexible. These techniques are particularly useful if you need to show the same data in several ways and you want to let the user decide how the data should be ordered.

Sorting

The GridView sorting features allow the user to reorder the results in the GridView by clicking a column header. It's convenient—and easy to implement.

Although you may not realize it, when you bind to a DataTable, you actually use another object called the DataView. The DataView sits between the ASP.NET web page binding and your DataTable. Usually it does little aside from providing the information from the associated DataTable. However, you can customize the DataView so it applies its own sort order. That way, you can also customize the data that appear in the web page without needing to actually modify your data.

You can create a new DataView object by hand and bind the DataView directly to a data control such as the GridView. However, the GridView and SqlDataSource controls make it even easier. They provide several properties you can set to control sorting. Once you've configured these properties, the sorting is automatic, and you still won't need to write any code in your page class.

To enable sorting, you must set the GridView.AllowSorting property to true. Next, you need to define a SortExpression for each column that can be sorted. In theory, a sort expression takes the form used in the ORDER BY clause of a SQL query and can use any syntax that's understood by the data source control. In practice, you'll almost always use a single field name to sort the grid using the data in that column. For example, here's how you could define the ProductName column so it sorts by alphabetically ordering rows:

```
<asp:BoundField DataField = "ProductName" HeaderText = "Product Name"  
SortExpression = "ProductName" />
```

Note that if you don't want a column to be sort-enabled, you simply don't set its SortExpression property. Figure 16-10 shows an example with a grid that has sort expressions for all three columns and is currently sorted by product name.

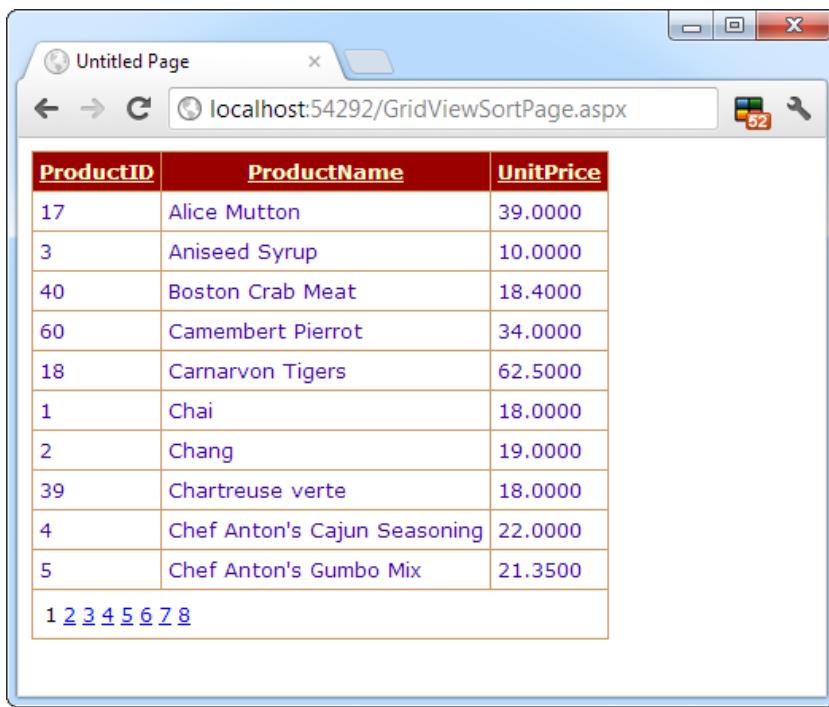


Figure 16-10. Sorting the GridView

Once you've associated a sort expression with the column and set the AllowSorting property to true, the GridView will render the headers with clickable links, as shown in Figure 16-10. However, it's up to the data source control to implement the actual sorting logic. How the sorting is implemented depends on the data source you're using.

Not all data sources support sorting, but the SqlDataSource does, provided the DataSourceMode property is set to DataSet (the default), not DataReader. In DataReader mode, the records are retrieved one at a time, and each record is stuffed into the bound control (such as a GridView) before the SqlDataSource moves to the next one. In DataSet mode, the entire results are placed in a DataSet and then the records are copied from the DataSet into the bound control. If the data need to be sorted, the sorting happens between these two steps—after the records are retrieved but before they're bound in the web page.

Note The sort is according to the data type of the column. numeric and date columns are ordered from largest to smallest. string columns are sorted alphanumerically without regard to case. Columns that contain binary data cannot be sorted. However, if you click a column header twice, the second click reverses the sort order, putting the records in descending order. (Click a third time to switch it back to ascending order.)

Sorting and Selecting

If you use sorting and selection at the same time, you'll discover another issue. To see this problem in action, select a row and then sort the data by any column. You'll see that the selection will remain, but it will shift to a

new item that has the same index as the previous item. In other words, if you select the second row and perform a sort, the second row will still be selected in the new page, even though this isn't the record you selected.

To fix this problem, you must simply set the `GridView.EnablePersistedSelection` property true. Now, ASP.NET will ensure that the selected item is identified by its data key. As a result, the selected item will remain selected, even if it moves to a new position in the `GridView` after a sort.

Paging

Often, a database search will return too many rows to be realistically displayed in a single page. If the client is using a slow connection, an extremely large `GridView` can take a frustrating amount of time to arrive. Once the data are retrieved, the user may find out they don't contain the right content anyway or that the search was too broad and they can't easily wade through all the results to find the important information.

The `GridView` handles this scenario with an automatic paging feature. When you use automatic paging, the full results are retrieved from the data source and placed into a `DataSet`. Once the `DataSet` is bound to the `GridView`, however, the data are subdivided into smaller groupings (for example, with 20 rows each), and only a single batch is sent to the user. The other groups are abandoned when the page finishes processing. When the user moves to the next page, the same process is repeated—in other words, the full query is performed once again. The `GridView` extracts just one group of rows, and the page is rendered.

To allow the user to skip from one page to another, the `GridView` displays a group of pager controls at the bottom of the grid. These pager controls could be previous/next links (often displayed as < and >) or number links (1, 2, 3, 4, 5, ...) that lead to specific pages. If you've ever used a search engine, you've seen paging at work.

By setting a few properties, you can make the `GridView` control manage the paging for you. Table 16-6 describes the key properties.

Table 16-6. Paging Members of the `GridView`

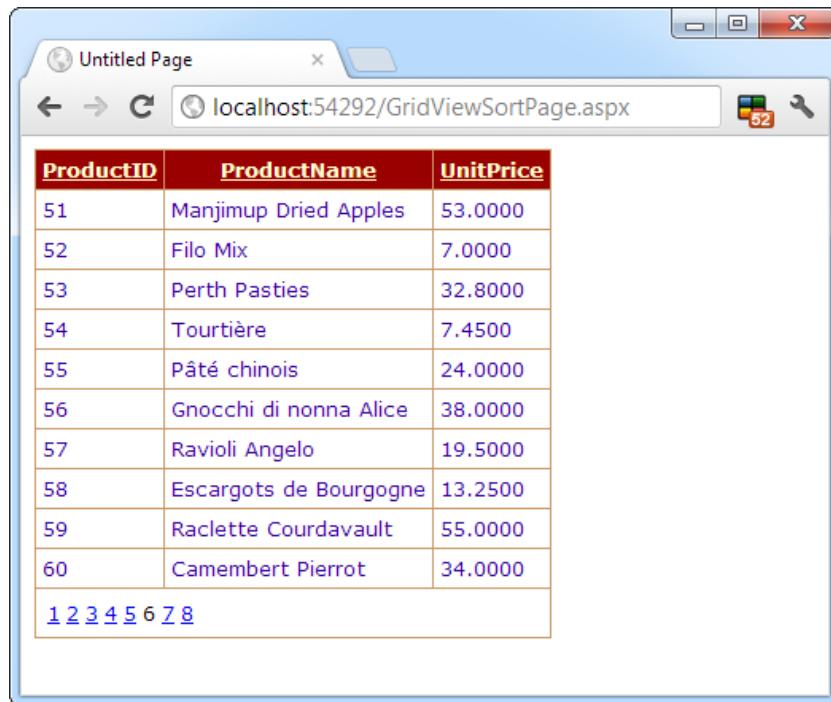
Property	Description
<code>AllowPaging</code>	Enables or disables the paging of the bound records. It is false by default.
<code>PageSize</code>	Gets or sets the number of items to display on a single page of the grid. The default value is 10.
<code>PageIndex</code>	Gets or sets the zero-based index of the currently displayed page, if paging is enabled.
<code>PagerSettings</code>	Provides a <code>PagerSettings</code> object that wraps a variety of formatting options for the pager controls. These options determine where the paging controls are shown and what text or images they contain. You can set these properties to fine-tune the appearance of the pager controls, or you can use the defaults.
<code>PagerStyle</code>	Provides a style object you can use to configure fonts, colors, and text alignment for the paging controls.
<code>PageIndexChanging</code> and <code>PageIndexChanged</code> events	Occur when one of the page selection elements is clicked, just before the <code>PageIndex</code> is changed (<code>PageIndexChanging</code>) and just after (<code>PageIndexChanged</code>).

To use automatic paging, you need to set `AllowPaging` to true (which shows the page controls), and you need to set `PageSize` to determine how many rows are allowed on each page.

Here's an example of a GridView control declaration that sets these properties:

```
<asp:GridView ID = "GridView1" runat = "server" DataSourceID = "sourceProducts"
    PageSize = "10" AllowPaging = "True" . . .>
    ...
</asp:GridView>
```

This is enough to start using paging. Figure 16-11 shows an example with ten records per page (for a total of eight pages).



The screenshot shows a web browser window titled "Untitled Page" with the URL "localhost:54292/GridviewSortPage.aspx". The main content area displays a GridView control containing 10 rows of product information. The columns are labeled "ProductID", "ProductName", and "UnitPrice". The data includes products like Manjimup Dried Apples, Filo Mix, Perth Pasties, Tourtière, Pâté chinois, Gnocchi di nonna Alice, Ravioli Angelo, Escargots de Bourgogne, Raclette Courdavault, and Camembert Pierrot. At the bottom of the grid, there is a navigation bar with links labeled 1, 2, 3, 4, 5, 6, 7, 8.

<u>ProductID</u>	<u>ProductName</u>	<u>UnitPrice</u>
51	Manjimup Dried Apples	53.0000
52	Filo Mix	7.0000
53	Perth Pasties	32.8000
54	Tourtière	7.4500
55	Pâté chinois	24.0000
56	Gnocchi di nonna Alice	38.0000
57	Ravioli Angelo	19.5000
58	Escargots de Bourgogne	13.2500
59	Raclette Courdavault	55.0000
60	Camembert Pierrot	34.0000

Figure 16-11. Paging the GridView

Paging and Selection

By default, paging and selection don't play nicely together. If you enable both for the GridView, you'll notice that the same row position remains selected as you move from one page to another. For example, if you select the first row on the first page and then move to the second page, the first row on the second page will become selected. To fix this quirk, set the `GridView.EnablePersistedSelection` property to true. Now, as you move from one page to another, the selection will automatically be removed from the GridView (and the `SelectedIndex` property will be set to -1). But if you move back to the page that held the originally selected row, that row will be re-selected. This behavior is intuitive, and it neatly ensures that your code won't be confused by a selected row that isn't currently visible.

paGING aND perFORMaNce

When you use paging, every time a new page is requested, the full Dataset is queried from the database. This means paging does not reduce the amount of time required to query the database. In fact, because the information is split into multiple pages and you need to repeat the query every time the user moves to a different page, the database load actually *increases*. However, because any given page contains only a subset of the total data, the page size is smaller and will be transmitted faster, reducing the client's wait. The end result is a more responsive and manageable page.

You can use paging in certain ways without increasing the amount of work the database needs to perform. One option is to cache the entire Dataset in server memory. That way, every time the user moves to a different page, you simply need to retrieve the data from memory and rebind it, avoiding the database query altogether. You'll learn how to use this technique in Chapter 23.

Using GridView Templates

So far, the examples have used the GridView control to show data using separate bound columns for each field. If you want to place multiple values in the same cell, or you want the unlimited ability to customize the content in a cell by adding HTML tags and server controls, you need to use a TemplateField.

The TemplateField allows you to define a completely customized *template* for a column. Inside the template you can add control tags, arbitrary HTML elements, and data binding expressions. You have complete freedom to arrange everything the way you want.

For example, imagine you want to create a column that combines the in-stock, on-order, and reorder level information for a product. To accomplish this trick, you can construct an ItemTemplate like this:

```
<asp:TemplateField HeaderText = "Status">
  <ItemTemplate>
    <b> In Stock:</b>
    <%# Eval("UnitsInStock") %> <br />
    <b> On Order:</b>
    <%# Eval("UnitsOnOrder") %> <br />
    <b> Reorder:</b>
    <%# Eval("ReorderLevel") %>
  </ItemTemplate>
</asp:TemplateField>
```

Note Your template only has access to the fields that are in the bound data object. so if you want to show the UnitsInStock, UnitsOnOrder, and ReorderLevel fields, you need to make sure the sqlDatasource query returns that information.

To create the data binding expressions, the template uses the `Eval()` method, which is a static method of the `System.Web.UI.DataBinder` class. `Eval()` is an indispensable convenience: it automatically retrieves the data item that's bound to the current row, uses reflection to find the matching field, and retrieves the value.

Tip The `Eval()` method also adds the extremely useful ability to format data fields on the fly. To use this you must call the overloaded version of the `Eval()` method that accepts an additional format string parameter. For example:

```
<%# Eval("BirthDate", "{0:MM/dd/yy}") %>
```

You can use any of the format strings defined in Table 16-3 and Table 16-4 with the `Eval()` method.

You'll notice that this example template includes three data binding expressions. These expressions get the actual information from the current row. The rest of the content in the template defines static text, tags, and controls.

You also need to make sure the data source provides these three pieces of information. If you attempt to bind a field that isn't present in your result set, you'll receive a runtime error. If you retrieve additional fields that are never bound to any template, no problem will occur.

Here's the revised data source with these fields:

```
<asp:SqlDataSource ID = "sourceProducts" runat = "server"  
ConnectionString = "<%$ ConnectionStrings:Northwind %>"  
SelectCommand = "SELECT ProductID, ProductName, UnitPrice, UnitsInStock,  
UnitsOnOrder,ReorderLevel FROM Products"  
UpdateCommand = "UPDATE Products SET ProductName = @ProductName,  
UnitPrice = @UnitPrice WHERE ProductID = @ProductID">  
</asp:SqlDataSource>
```

When you bind the `GridView`, it fetches the data from the data source and walks through the collection of items. It processes the `ItemTemplate` for each item, evaluates the data binding expressions, and adds the rendered HTML to the table. You're free to mix template columns with other column types. Figure 16-12 shows an example with several normal columns and the template column at the end.

ID	Product Name	Price	Status
1	Chai	18.0000	In Stock: 39 On Order: 0 Reorder: 10
2	Chang	19.0000	In Stock: 17 On Order: 40 Reorder: 25
3	Aniseed Syrup	10.0000	In Stock: 13 On Order: 70 Reorder: 25
4	Chef Anton's Cajun Seasoning	22.0000	In Stock: 53 On Order: 0 Reorder: 0
5	Chef Anton's Gumbo Mix	21.3500	In Stock: 0 On Order: 0 Reorder: 0
6	Grandma's Boysenberry Spread	25.0000	In Stock: 120 On Order: 0 Reorder: 25
7	Uncle Bob's Organic Dried Pears	30.0000	In Stock: 15 On Order: 0 Reorder: 10

Figure 16-12. A GridView with a template column

Using Multiple Templates

The previous example uses a single template to configure the appearance of data items. However, the ItemTemplate isn't the only template that the TemplateField provides. In fact, the TemplateField allows you to configure various aspects of its appearance with a number of templates. Inside every template column, you can use the templates listed in Table 16-7.

Table 16-7. *TemplateField Templates*

Mode	Description
HeaderTemplate	Determines the appearance and content of the header cell.
FooterTemplate	Determines the appearance and content of the footer cell (if you set ShowFooter to true).
ItemTemplate	Determines the appearance and content of each data cell.
AlternatingItemTemplate	Determines the appearance and content of even-numbered rows. For example, if you set the AlternatingItemTemplate to have a shaded background color, the GridView applies this shading to every second row.
EditItemTemplate	Determines the appearance and controls used in edit mode.
InsertItemTemplate	Determines the appearance and controls used in edit mode. The GridView doesn't support this template, but the DetailsView and FormView controls (which are described later in this chapter) do.

Of the templates listed in Table 16-7, the EditItemTemplate is one of the most useful because it gives you the ability to control the editing experience for the field. If you don't use template fields, you're limited to ordinary text boxes, and you won't have any validation. The GridView also defines two templates you can use outside any column. These are the PagerTemplate, which lets you customize the appearance of pager controls, and the EmptyDataTemplate, which lets you set the content that should appear if the GridView is bound to an empty data object.

Editing Templates in Visual Studio

Visual Studio includes solid support for editing templates in the web page designer. To try this, follow these steps:

1. Create a GridView with at least one template column.
2. Select the GridView, and click Edit Templates in the smart tag. This switches the GridView into template edit mode.
3. In the smart tag, use the Display drop-down list to choose the template you want to edit (see Figure 16-13). You can choose either of the two templates that apply to the whole GridView (EmptyDataTemplate or PagerTemplate), or you can choose a specific template for one of the template columns.

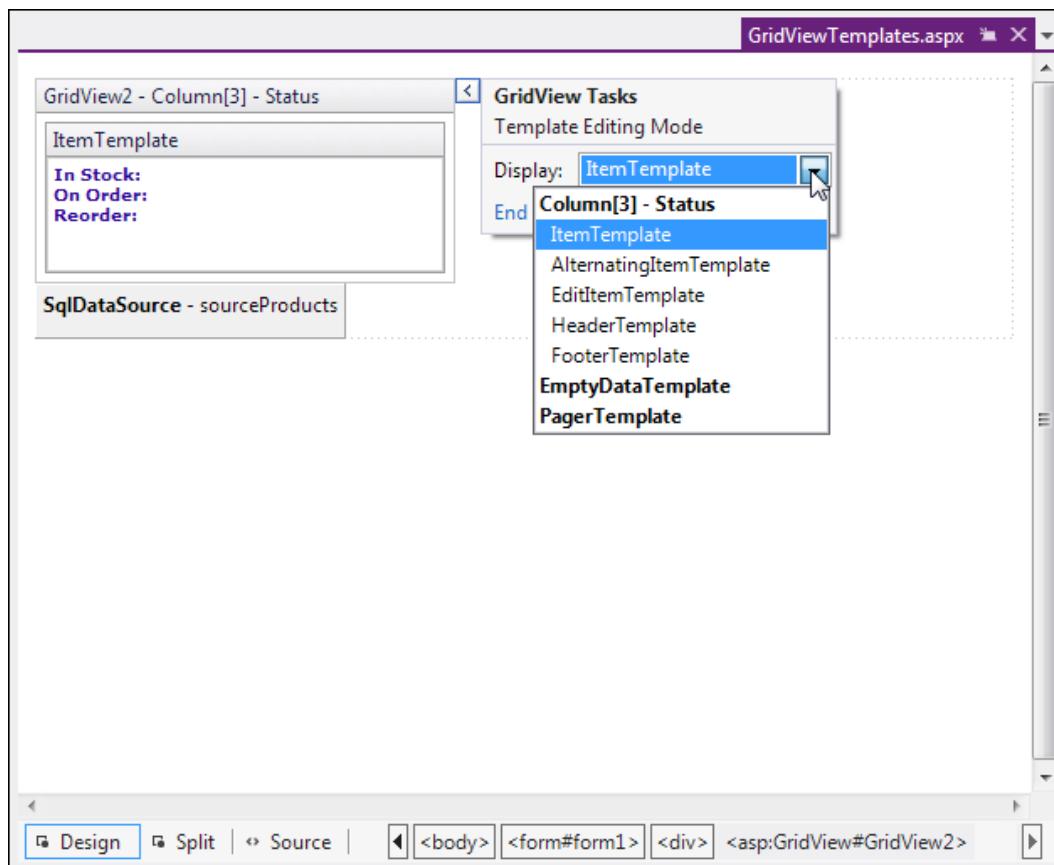


Figure 16-13. Editing a template in Visual Studio

4. Enter your content in the control. You can enter static content, drag-and-drop controls, and so on.
5. When you're finished, choose End Template Editing from the smart tag.

Handling Events in a Template

In some cases, you might need to handle events that are raised by the controls you add to a template column. For example, imagine you want to add a clickable image link by including an ImageButton control. This is easy enough to accomplish:

```
<asp:TemplateField HeaderText = "Status">
<ItemTemplate>
  <asp:ImageButton ID = "ImageButton1" runat = "server"
    ImageUrl = "statuspic.gif" />
</ItemTemplate>
</asp:TemplateField>
```

The problem is that if you add a control to a template, the GridView creates multiple copies of that control, one for each data item. When the ImageButton is clicked, you need a way to determine which image was clicked and to which row it belongs.

The way to resolve this problem is to use an event from the GridView, *not* the contained button. The GridView.RowCommand event serves this purpose because it fires whenever any button is clicked in any template. This process, where a control event in a template is turned into an event in the containing control, is called *event bubbling*.

Of course, you still need a way to pass information to the RowCommand event to identify the row where the action took place. The secret lies in two string properties that all button controls provide: CommandName and CommandArgument. CommandName sets a descriptive name you can use to distinguish clicks on your ImageButton from clicks on other button controls in the GridView. The CommandArgument supplies a piece of row-specific data you can use to identify the row that was clicked. You can supply this information using a data binding expression.

Here's a template field that contains the revised ImageButton tag:

```
<asp:TemplateField HeaderText = "Status">
<ItemTemplate>
  <asp:ImageButton ID = "ImageButton1" runat = "server"
    ImageUrl = "statuspic.gif"
    CommandName = "StatusClick" CommandArgument = '<%# Eval("ProductID") %>' />
</ItemTemplate>
</asp:TemplateField>
```

And here's the code you need in order to respond when an ImageButton is clicked:

```
protected void GridView1_RowCommand(object sender, GridViewCommandEventArgs e)
{
  if (e.CommandName == "StatusClick")
    lblInfo.Text = "You clicked product #" + e.CommandArgument.ToString();
}
```

This example displays a simple message with the ProductID in a label.

Editing with a Template

One of the best reasons to use a template is to provide a better editing experience. In the previous chapter, you saw how the GridView provides automatic editing capabilities—all you need to do is switch a row into edit mode by setting the GridView.EditIndex property. The easiest way to make this possible is to add a CommandField column with the ShowEditButton set to true. Then, the user simply clicks a link in the appropriate row to begin editing it. At this point, every label in every column is replaced by a text box (unless the field is read-only).

The standard editing support has several limitations:

It's not always appropriate to edit values using a text box: Certain types of data are best handled with other controls (such as drop-down lists). Large fields need multiline text boxes, and so on.

You get no validation: It would be nice to restrict the editing possibilities so that currency figures can't be entered as negative numbers, for example. You can do that by adding validator controls to an EditItemTemplate.

The visual appearance is often ugly: A row of text boxes across a grid takes up too much space and rarely seems professional.

In a template column, you don't have these issues. Instead, you explicitly define the edit controls and their layout using the `EditItemTemplate`. This can be a somewhat laborious process.

Here's the template column used earlier for stock information with an editing template:

```
<asp:TemplateField HeaderText = "Status">
<ItemStyle Width = "100px" />
<ItemTemplate>
  <b> In Stock:</b> <%# Eval("UnitsInStock") %> <br />
  <b> On Order:</b> <%# Eval("UnitsOnOrder") %> <br />
  <b> Reorder:</b> <%# Eval("ReorderLevel") %>
</ItemTemplate>
<EditItemTemplate>
  <b> In Stock:</b> <%# Eval("UnitsInStock") %> <br />
  <b> On Order:</b> <%# Eval("UnitsOnOrder") %> <br /> <br />
  <b> Reorder:</b>
  <asp:TextBox Text = '<%# Bind("ReorderLevel") %>' Width = "25px"
    runat = "server" id = "txtReorder" />
</EditItemTemplate>
</asp:TemplateField>
```

Figure 16-14 shows the row in edit mode.

	ID	Product Name	Price	Status
Update Cancel	1	Chai	18.0000	In Stock: 39 On Order: 0 Reorder: 10
Edit	2	Chang	19.0000	In Stock: 17 On Order: 40 Reorder: 25
Edit	3	Aniseed Syrup	10.0000	In Stock: 13 On Order: 70 Reorder: 25
Edit	4	Chef Anton's Cajun Seasoning	22.0000	In Stock: 53 On Order: 0 Reorder: 0
Edit	5	Chef Anton's Gumbo Mix	21.3500	In Stock: 0 On Order: 0 Reorder: 0

Figure 16-14. Using an edit template

When binding an editable value to a control, you must use the `Bind()` method in your data binding expression instead of the ordinary `Eval()` method. Unlike the `Eval()` method, which can be placed anywhere in a page, the `Bind()` method must be used to set a control property. Only the `Bind()` method creates the two-way link, ensuring that updated values will be returned to the server.

One interesting detail here is that even though the item template shows three fields, the editing template allows only one of these to be changed. When the GridView commits an update, it will submit only the bound, editable parameters. In the previous example, this means the GridView will pass back a @ReorderLevel parameter but *not* a @UnitsInStock or @UnitsOnOrder parameter. This is important because when you write your parameterized update command, it must use only the parameters you have available. Here's the modified SqlDataSource control with the correct command:

```
<asp:SqlDataSource ID = "sourceProducts" runat = "server"
    ConnectionString = "<%$ ConnectionStrings:Northwind %>" 
    SelectCommand = "SELECT ProductID, ProductName, UnitPrice, UnitsInStock,
        UnitsOnOrder, ReorderLevel FROM Products"
    UpdateCommand = "UPDATE Products SET ProductName = @ProductName, UnitPrice = @UnitPrice,
        ReorderLevel = @ReorderLevel WHERE ProductID = @ProductID">
</asp:SqlDataSource>
```

Editing with Validation

Now that you have your template ready, why not add a frill, such as a validator, to catch editing mistakes? In the following example, a RangeValidator prevents changes that put the ReorderLevel at less than 0 or more than 100:

```
<asp:TemplateField HeaderText = "Status">
    <ItemStyle Width = "100px" />
    <ItemTemplate>
        <b> In Stock:</b> <%# Eval("UnitsInStock") %> <br />
        <b> On Order:</b> <%# Eval("UnitsOnOrder") %> <br />
        <b> Reorder:</b> <%# Eval("ReorderLevel") %>
    </ItemTemplate>
    <EditItemTemplate>
        <b> In Stock:</b> <%# Eval("UnitsInStock") %> <br />
        <b> On Order:</b> <%# Eval("UnitsOnOrder") %> <br /> <br />
        <b> Reorder:</b>
        <asp:TextBox Text = '<%# Bind("ReorderLevel") %>' Width = "25px"
            runat = "server" id = "txtReorder" />
        <asp:RangeValidator id = "rngValidator" MinimumValue = "0" MaximumValue = "100"
            ControlToValidate = "txtReorder" runat = "server"
            ErrorMessage = "Value out of range." Type = "Integer"/>
    </EditItemTemplate>
</asp:TemplateField>
```

Figure 16-15 shows the validation at work. If the value isn't valid, the browser doesn't allow the page to be posted back, and no database code runs.

The screenshot shows a Microsoft Internet Explorer window titled "Untitled Page" with the URL "localhost:54292/GridviewTemplates.aspx". The page displays a GridView containing product information. The columns are labeled "ID", "Product Name", "Price", and "Status". Row 1 (ID 1) has an "Update" link and a "Cancel" link. The "Product Name" is "Chai", "Price" is "18.0000", and "Status" shows "In Stock: 39", "On Order: 0", and a "Reorder" input field set to "-5" with a tooltip "Value out of range.". Row 2 (ID 2) has an "Edit" link. The "Product Name" is "Chang", "Price" is "19.0000", and "Status" shows "In Stock: 17", "On Order: 40", and "Reorder: 25". Row 3 (ID 3) has an "Edit" link. The "Product Name" is "Aniseed Syrup", "Price" is "10.0000", and "Status" shows "In Stock: 13", "On Order: 70", and "Reorder: 25". Row 4 (ID 4) has an "Edit" link. The "Product Name" is "Chef Anton's Cajun Seasoning", "Price" is "22.0000", and "Status" shows "In Stock: 53", "On Order: 0", and "Reorder: 0". Row 5 (ID 5) has an "Edit" link. The "Product Name" is "Lumber", "Price" is "36.0000", and "Status" shows "In Stock: 0". A vertical scrollbar is visible on the right side of the grid.

	ID	Product Name	Price	Status
Update Cancel	1	Chai	18.0000	In Stock: 39 On Order: 0 Reorder: -5 Value out of range.
Edit	2	Chang	19.0000	In Stock: 17 On Order: 40 Reorder: 25
Edit	3	Aniseed Syrup	10.0000	In Stock: 13 On Order: 70 Reorder: 25
Edit	4	Chef Anton's Cajun Seasoning	22.0000	In Stock: 53 On Order: 0 Reorder: 0
	5	Lumber	36.0000	In Stock: 0

Figure 16-15. Creating an edit template with validation

Note The `sqlDataSource` is intelligent enough to handle validation properly even if you have disabled client-side validation (or the browser doesn't support it). In this situation, the page is posted back, but the `sqlDataSource` notices that it contains invalid data (by inspecting the `Page.IsValid` property), and doesn't attempt to perform the update. For more information about client-side and server-side validation, refer to Chapter 9.

Editing Without a Command Column

So far, all the examples you've seen have used a `CommandField` that automatically generates edit controls. However, now that you've made the transition to a template-based approach, it's worth considering how you can add your own edit controls.

It's actually quite easy. All you need to do is add a button control to the item template and set the `CommandName` to `Edit`. This automatically triggers the editing process, which fires the appropriate events and switches the row into edit mode.

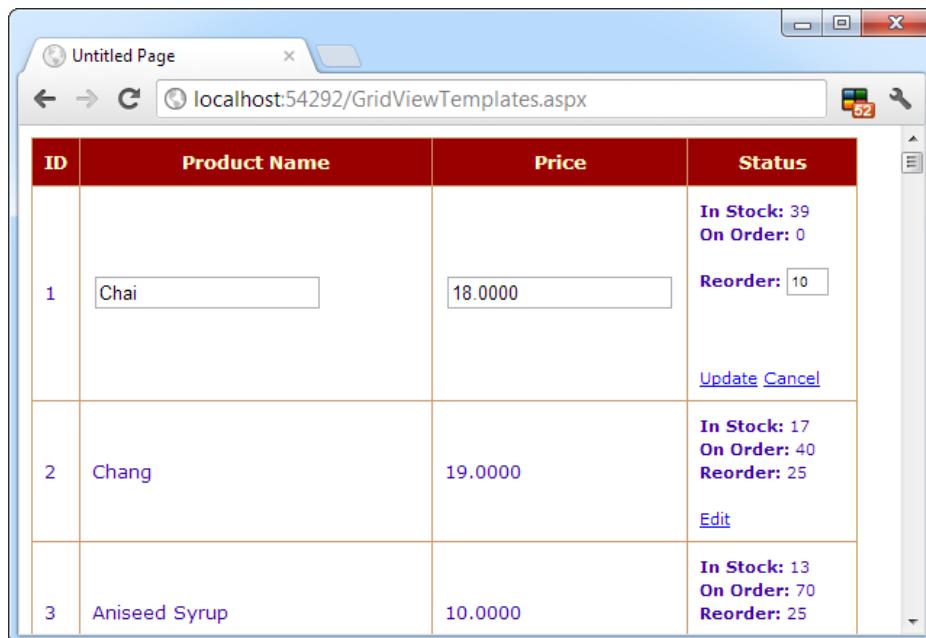
```
<ItemTemplate>
<b> In Stock:</b> <%# Eval("UnitsInStock") %> <br />
<b> On Order:</b> <%# Eval("UnitsOnOrder") %> <br />
<b> Reorder:</b> <%# Eval("ReorderLevel") %>
<br /> <br />
<asp:LinkButton runat="server" Text="Edit"
  CommandName="Edit" ID="LinkButton1" />
</ItemTemplate>
```

In the edit item template, you need two more buttons with CommandName values of Update and Cancel:

```
<EditItemTemplate>
<b> In Stock:</b> <%# Eval("UnitsInStock") %> <br />
<b> On Order:</b> <%# Eval("UnitsOnOrder") %> <br /> <br />
<b> Reorder:</b>
<asp:TextBox Text = '<%# Bind("ReorderLevel") %>' Width = "25px"
runat = "server" id = "txtReorder" />
<br /> <br />
<asp:LinkButton runat = "server" Text = "Update"
CommandName = "Update" ID = "LinkButton1" />
<asp:LinkButton runat = "server" Text = "Cancel"
CommandName = "Cancel" ID = "LinkButton2" CausesValidation = "False" />
</EditItemTemplate>
```

Notice that the Cancel button must have its CausesValidation property set to false to bypass validation. That way, you can cancel the edit even if the current data aren't valid.

As long as you use these names, the GridView editing events will fire and the data source controls will react in the same way as if you were using the automatically generated editing controls. Figure 16-16 shows the custom edit buttons.



The screenshot shows a Microsoft Internet Explorer window titled 'Untitled Page' with the URL 'localhost:54292/GridViewTemplates.aspx'. The page displays a GridView with four columns: 'ID', 'Product Name', 'Price', and 'Status'. The 'Status' column contains status information and edit links. Row 1 (ID 1) has 'Chai' in the Product Name column and '18.0000' in the Price column. The Status column shows 'In Stock: 39' and 'On Order: 0'. Below this, there is a 'Reorder:' label followed by a text input box containing '10', and a row of blue underlined links: 'Update' and 'Cancel'. Row 2 (ID 2) has 'Chang' in the Product Name column and '19.0000' in the Price column. The Status column shows 'In Stock: 17', 'On Order: 40', and 'Reorder: 25'. Below this, there is a blue underlined link 'Edit'. Row 3 (ID 3) has 'Aniseed Syrup' in the Product Name column and '10.0000' in the Price column. The Status column shows 'In Stock: 13', 'On Order: 70', and 'Reorder: 25'.

ID	Product Name	Price	Status
1	Chai	18.0000	In Stock: 39 On Order: 0 Reorder: 10 Update Cancel
2	Chang	19.0000	In Stock: 17 On Order: 40 Reorder: 25 Edit
3	Aniseed Syrup	10.0000	In Stock: 13 On Order: 70 Reorder: 25

Figure 16-16. Custom edit controls

The DetailsView and FormView

The GridView excels at showing a dense table with multiple rows of information. However, sometimes you want to provide a detailed look at a single record. You could work out a solution using a template column in a GridView, but ASP.NET includes two controls that are tailored for this purpose: the DetailsView and the

FormView. Both show a single record at a time but can include optional pager buttons that let you step through a series of records (showing one per page). Both give you an easy way to insert a new record, which the GridView doesn't allow. And both support templates, but the FormView *requires* them. This is the key distinction between the two controls.

One other difference is the fact that the DetailsView renders its content inside a table, while the FormView gives you the flexibility to display your content without a table. Thus, if you're planning to use templates, the FormView gives you the most flexibility. But if you want to avoid the complexity of templates, the DetailsView gives you a simpler model that lets you build a multirow data display out of field objects, in much the same way that the GridView is built out of column objects.

Now that you understand the features of the GridView, you can get up to speed with the DetailsView and the FormView quite quickly. That's because both borrow a portion of the GridView model.

The DetailsView

The DetailsView displays a single record at a time. It places each field in a separate row of a table.

You saw in Chapter 15 how to create a basic DetailsView to show the currently selected record. The DetailsView also allows you to move from one record to the next using paging controls, if you've set the AllowPaging property to true. You can configure the paging controls using the PagerStyle and PagerSettings properties in the same way as you tweak the pager for the GridView.

Figure 16-17 shows the DetailsView when it's bound to a set of product records, with full product information.

The screenshot shows a Windows application window titled "Untitled Page". The address bar indicates the URL is "localhost:54292/DetailsViewTest.aspx". The main content area displays a DetailsView control containing a table of product information. The table has the following data:

ProductID	1
ProductName	Chai
SupplierID	1
CategoryID	1
QuantityPerUnit	10 boxes x 20 bags
UnitPrice	18.0000
UnitsInStock	39
UnitsOnOrder	0
ReorderLevel	10
Discontinued	<input type="checkbox"/>

Below the table, there are three buttons: Edit, Delete, and New. At the very bottom of the DetailsView control, there is a pager with links labeled 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ..., indicating multiple pages of data.

Figure 16-17. The DetailsView with paging

It's tempting to use the DetailsView pager controls to make a handy record browser. Unfortunately, this approach can be quite inefficient. One problem is that a separate postback is required each time the user moves from one record to another (whereas a grid control can show multiple records on the same page). But the real drawback is that each time the page is posted back, the full set of records is retrieved, even though only a single record is shown. This results in needless extra work for the database server. If you choose to implement a record browser page with the DetailsView, at a bare minimum you must enable caching to reduce the database work (see Chapter 23).

Tip It's almost always a better idea to use another control to let the user choose a specific record (for example by choosing an ID from a list box), and then show the full record in the DetailsView using a parameterized query that matches just the selected record. Chapter 15 demonstrates this technique.

Defining Fields

The DetailsView uses reflection to generate the fields it shows. This means it examines the data object and creates a separate row for each field it finds, just like the GridView. You can disable this automatic row generation by setting AutoGenerateRows to false. It's then up to you to declare information you want to display.

Interestingly, you use the same field tags to build a DetailsView as you use to design a GridView. For example, fields from the data item are represented with the BoundField tag, buttons can be created with the ButtonField, and so on. For the full list, refer to the earlier Table 16-1.

The following code defines a DetailsView that shows product information. This tag creates the same grid of information shown in Figure 16-17, when AutoGenerateRows was set to true.

```
<asp:DetailsView ID = "DetailsView1" runat = "server" AutoGenerateRows = "False"
DataSourceID = "sourceProducts">
<Fields>
  <asp:BoundField DataField = "ProductID" HeaderText = "ProductID"
    ReadOnly = "True" />
  <asp:BoundField DataField = "ProductName" HeaderText = "ProductName" />
  <asp:BoundField DataField = "SupplierID" HeaderText = "SupplierID" />
  <asp:BoundField DataField = "CategoryID" HeaderText = "CategoryID" />
  <asp:BoundField DataField = "QuantityPerUnit" HeaderText = "QuantityPerUnit" />
  <asp:BoundField DataField = "UnitPrice" HeaderText = "UnitPrice" />
  <asp:BoundField DataField = "UnitsInStock" HeaderText = "UnitsInStock" />
  <asp:BoundField DataField = "UnitsOnOrder" HeaderText = "UnitsOnOrder" />
  <asp:BoundField DataField = "ReorderLevel" HeaderText = "ReorderLevel" />
  <asp:CheckBoxField DataField = "Discontinued" HeaderText = "Discontinued" />
</Fields>
...
</asp:DetailsView>
```

You can use the BoundField tag to set properties such as header text, formatting string, editing behavior, and so on (refer to Table 16-2). In addition, you can use the ShowHeader property. When it's false, the header text is left out of the row, and the field data take up both cells.

Tip Rather than coding each field by hand, you can use the same shortcut as you used with the GridView—simply select the control at design time, and select Refresh schema from the smart tag.

The field model isn't the only part of the GridView that the DetailsView control adopts. It also uses a similar set of styles, a similar set of events, and a similar editing model. The only difference is that instead of creating a dedicated column for editing controls, you simply set one of the Boolean properties of the DetailsView, such as AutoGenerateDeleteButton, AutoGenerateEditButton, and AutoGenerateInsertButton. The links for these tasks are added to the bottom of the DetailsView. When you add or edit a record, the DetailsView uses standard text box controls (see Figure 16-18), just as the GridView does. For more editing flexibility, you'll want to use templates with the DetailsView (by adding a TemplateField instead of a BoundField) or the FormView control (as described next).

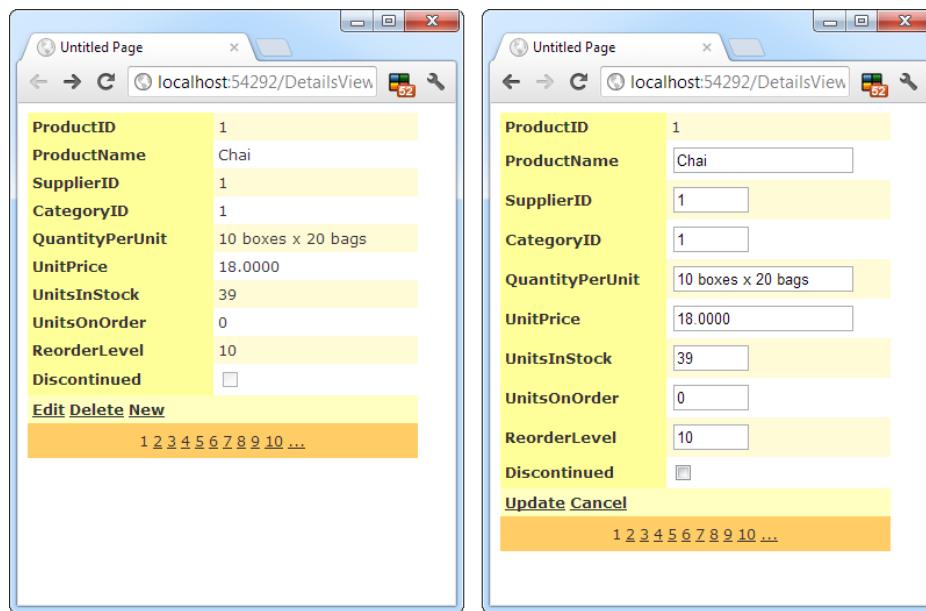


Figure 16-18. Editing in the DetailsView

The FormView

If you need the ultimate flexibility of templates, the FormView provides a template-only control for displaying and editing a single record.

The beauty of the FormView template model is that it matches quite closely the model of the TemplateField in the GridView. This means you can work with the following templates:

- ItemTemplate
- EditItemTemplate
- InsertItemTemplate
- FooterTemplate
- HeaderTemplate
- EmptyDataTemplate
- PagerTemplate

Note Unlike the GridView and DetailsView, which allow you to add as many TemplateField objects as you want, the FormView allows just a single copy of each template. If you want to show multiple values, you must add binding expressions to the same ItemTemplate.

You can use the same template content you use with a TemplateField in a GridView in the FormView. Earlier in this chapter, you saw how you can use a template field to combine the stock information of a product into one column (as shown in Figure 16-12). Here's how you can use the same template in the FormView:

```
<asp:FormView ID = "FormView1" runat = "server" DataSourceID = "sourceProducts">
<ItemTemplate>
<b> In Stock:</b>
<%# Eval("UnitsInStock") %>
<br />
<b> On Order:</b>
<%# Eval("UnitsOnOrder") %>
<br />
<b> Reorder:</b>
<%# Eval("ReorderLevel") %>
<br />
</ItemTemplate>
</asp:FormView>
```

Like the DetailsView, the FormView can show a single record at a time. (If the data source has more than one record, you'll see only the first one.) You can deal with this issue by setting the AllowPaging property to true so that paging links are automatically created. These links allow the user to move from one record to the next, as in the previous example with the DetailsView.

Another option is to bind to a data source that returns just one record. Figure 16-19 shows an example where a drop-down list control lets you choose a product, and a second data source shows the matching record in the FormView control.

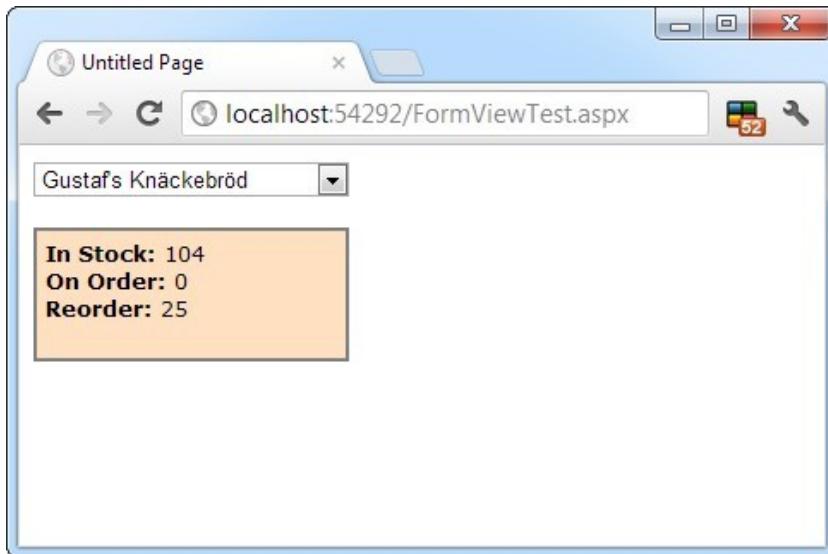


Figure 16-19. A FormView that shows a single record

Here's the markup you need to define the drop-down list and its data source:

```
<asp:SqlDataSource ID = "sourceProducts" runat = "server"
    ConnectionString = "<%$ ConnectionStrings:Northwind %>"
    SelectCommand = "SELECT ProductID, ProductName FROM Products">
</asp:SqlDataSource>

<asp:DropDownList ID = "lstProducts" runat = "server"
    AutoPostBack = "True" DataSourceID = "sourceProducts"
    DataTextField = "ProductName" DataValueField = "ProductID" Width = "184px">
</asp:DropDownList>
```

The FormView uses the template from the previous example (it's the shaded region on the page). Here's the markup for the FormView (not including the template) and the data source that gets the full details for the selected product.

```
<asp:SqlDataSource ID = "sourceProductFull" runat = "server"
    ConnectionString = "<%$ ConnectionStrings:Northwind %>"
    SelectCommand = "SELECT * FROM Products WHERE ProductID = @ProductID">
    <SelectParameters>
        <asp:ControlParameter Name = "ProductID"
            ControlID = "lstProducts" PropertyName = "SelectedValue" />
    </SelectParameters>
</asp:SqlDataSource>

<asp:FormView ID = "formProductDetails" runat = "server"
    DataSourceID = "sourceProductFull"
    BackColor = "#FFE0C0" CellPadding = "5">
    <ItemTemplate>
        ...
    </ItemTemplate>
</asp:FormView>
```

Note If you want to support editing with the FormView, you need to add button controls that trigger update processes, as described in the "Editing with a Template" section.

The Last Word

In this chapter, you considered everything you need to build rich data-bound pages. You took a detailed tour of the GridView and considered its support for formatting, selecting, sorting, paging, using templates, and editing. You also considered the DetailsView and the FormView, which allow you to display and edit individual records. Using these three controls, you can build all-in-one pages that display and edit data without needing to write pages of ADO.NET code. Best of all, every data control is thoroughly configurable, which means you can tailor it to fit just about any web application.



Security Fundamentals

Ordinarily, your ASP.NET website is available to anyone who connects to your web server, whether over a local network or the Internet. Although this is ideal for many web applications (and it suits the original spirit of the Internet), it isn't always an appropriate design choice. For example, an e-commerce site needs to provide a secure shopping experience to attract customers. A subscription-based site needs to limit content to extract a fee. And even a wide-open public site may provide some resources or features that shouldn't be available to all users.

ASP.NET provides an extensive security model that makes it easy to protect your web applications. Although this security model is powerful and profoundly flexible, it can appear confusing because of the many different layers that it includes. Much of the work in securing your application isn't writing code but, rather, determining the appropriate places to implement your security strategy.

In this chapter, you'll sort out the tangled ASP.NET security model. You'll learn two ways to authenticate users—first in using *forms authentication* (which is ideal for a public website that uses a custom database) and then in using *Windows authentication* (which is ideal for an intranet application on a company network).

Understanding Security Requirements

The first step in securing your applications is deciding where you require security and what it should protect. For example, you may need to block access in order to protect private information. Or, maybe you just want to enforce a pay-for-content system. Perhaps you don't need any sort of security at all, but you want an optional login feature to provide personalization for frequent visitors. These requirements will determine the approach you use.

Security doesn't have to be complex, but it must be wide-ranging and multilayered. For example, consider an e-commerce website that allows users to view reports of their recently placed orders. You probably already know the first line of defense that this website should use: a login page that forces users to identify themselves before they can see any personal information. In this chapter, you'll learn how to use this sort of authentication system. However, it's important to realize that, on its own, this layer of protection is not enough to truly secure your system. You also need to protect the back-end database with a strong password, and you might even choose to encrypt sensitive information before you store it (scrambling it so that it's unreadable without the right key to decrypt it). Steps like these protect your website from attacks that get beyond your authentication system. For example, they can deter a disgruntled employee with an account on the local network, a hacker who has gained access to your network through the company firewall, or a careless technician who discards a hard drive used for data storage without erasing it first.

Furthermore, you'll need to hunt carefully for weaknesses in the code you've written. A surprising number of websites fall prey to relatively simple attacks in which a malicious user tampers with a query string argument or a bit of HTML in the rendered page. In the e-commerce example, you need to make sure that a user who successfully logs in can't view another user's recent orders. Imagine that you've created a ViewOrders.aspx page that takes a query string argument named userID, like this:

`http://localhost/InsecureStore/ViewOrders.aspx?userID=4191`

This example is a security nightmare because any user can easily modify the userID parameter by editing the URL to see another user's information. A better solution would be to design the ViewOrders.aspx page so that it gets the user ID from the currently logged-on user identity (a trick you'll learn in this chapter) and then uses that to construct the right database command.

Note Another example of a security vulnerability introduced by poor coding is the ever-common SQL injection attack. You learned in Chapter 14 to prevent this attack by using parameterized database commands.

When designing with security in mind, it's important to consider the different avenues for attack. However, you can't always anticipate potential problems. For that reason, it makes great sense to layer your security. The mantra of security architects can be summed up as follows: *Don't force an attacker to do one impossible thing to break into your system—force the attacker to do several.*

Testing and Deploying Security Settings

As you already know, live ASP.NET websites use the IIS web-hosting software. But when you begin building and testing web applications, you rely on the built-in Visual Studio web server instead. This difference has two effects on website security:

The website user account: When you run a website in Visual Studio, the test web server uses your Windows user account. As a result, your code is allowed to perform tasks (reading a file, writing to a log, connecting to a database, and so on) that might not be allowed on a live web server. To make sure everything keeps working when you deploy your web application, you need to know a bit more about how IIS and ASP.NET use Windows accounts—a topic you'll tackle in Chapter 26.

Security configuration: You configure the security settings differently, including the type of authentication your web application uses and the authorization rules that govern who can access different pages. In a test site, you can apply these settings in two ways: by editing the web.config file or by running the Website Administration Tool (WAT), which you first met in Chapter 5. In a deployed application, you can keep using the security settings you set during development, you can edit the web.config by hand, or you can use the convenient IIS Manager configuration tool.

In addition, there are some security settings that come into play only when you configure a deployed website. For example, you can use IIS Manager to switch on Secure Sockets Layer (SSL), an encryption standard that makes sure malicious hackers can't eavesdrop on the messages sent between the user and your web server. You can also configure the specific protocols that IIS uses for Windows authentication (in other words, the exact security standard that's used to transmit user information and log someone in).

This chapter explains how to apply security to a website during development. The concepts and details you'll learn will apply equally well to a deployed application. However, if you want to adjust your security configuration after you deploy your application, or if you want to get more advanced with SSL and Windows authentication settings, it's worth learning about IIS Manager. Chapter 26 has all the details.

Authentication and Authorization

Two concepts form the basis of any discussion about security:

Authentication: This is the process of determining users identities and forcing those users to prove they are who they claim to be. Usually, this involves entering credentials (typically a user name and password) into some sort of login page or window. These credentials are then authenticated against the Windows user accounts on a computer, a list of users in a file, or a back-end database.

Authorization: Once a user is authenticated, authorization is the process of determining whether that user has sufficient permission to perform a given action (such as viewing a page or retrieving information from a database). Windows imposes some authorization checks (for example, when you open a file), but your code will probably want to impose its own checks (for example, when a user performs a task in your web application such as submitting an order, assigning a project, or giving a promotion).

Authentication and authorization are the cornerstones of a secure user-based site. The Windows operating system provides a good analogy. When you first boot up your computer, you supply a user ID and password, thereby authenticating yourself to the system. After that step, every time you interact with a restricted resource (such as a file, database, registry key, and so on), Windows quietly performs authorization checks to ensure your user account has the necessary rights.

You can use two types of authentication to secure an ASP.NET website:

Forms authentication: With forms authentication, ASP.NET is in charge of authenticating users, tracking them, and authorizing every request. Usually, forms authentication works in conjunction with a database where you store user information (such as user names and passwords), but you have complete flexibility. You could even store user information in a plain text file or write customized login code that calls a remote service. Forms authentication is the best and most flexible way to run a subscription site or e-commerce store.

Windows authentication: With Windows authentication, the web server forces every user to log in as a Windows user. (Depending on the specific configuration you use, this login process may take place automatically, as it does in the Visual Studio test web server, or it may require that the user type a name and password into a login dialog box.) This system requires that all users have Windows user accounts on the server (although users could share accounts). This scenario is poorly suited for a public web application, but it is often ideal for an intranet or company-specific site designed to provide resources for a limited set of users.

You'll concentrate on these two approaches in this chapter. First, you'll explore the forms authentication model, then, you'll consider Windows authentication.

Forms Authentication

In old-fashioned ASP programming, developers had to create their own security systems. A common approach was to insert a little snippet of code at the beginning of every secure page. This code would check for the existence of a custom cookie. If the cookie didn't exist, the user would be redirected to a login page, where the cookie would be created after a successful login.

ASP.NET uses the same approach in its forms authentication model. You are still responsible for creating the login page (although you can use a set of specially designed controls to help you, as described in Chapter 20). However, you don't need to create the security cookie manually, or check for it in secure pages, because ASP.NET handles these tasks automatically. You also benefit from ASP.NET's support for sophisticated validation algorithms, which make it all but impossible for users to spoof their own cookies or try other hacking tricks to fool your application into giving them access.

Figure 19-1 shows a simplified security diagram of the forms authentication model in ASP.NET.

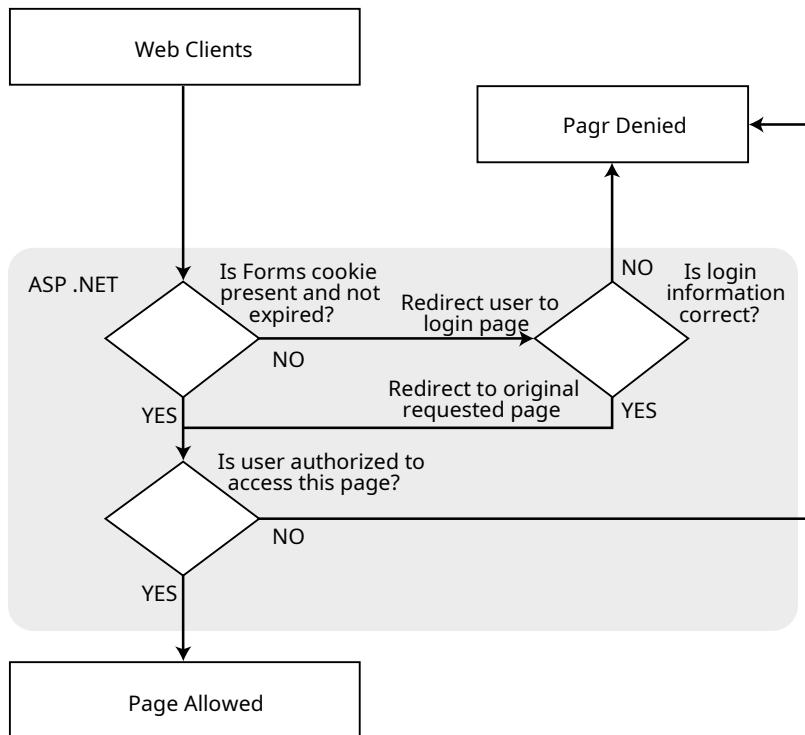


Figure 19-1. ASP.NET forms authentication

To implement forms-based security, follow these three steps:

1. Set the authentication mode to forms authentication in the web.config file. (If you prefer a graphical tool, you can use the WAT during development or IIS Manager after deployment.)
2. Restrict anonymous users from a specific page or directory in your application.
3. Create the login page.

You'll walk through these steps in the following sections.

Web.config Settings

You define the type of security in the web.config file by using the <authentication> tag.

The following example configures the application to use forms authentication by using the <authentication> tag. It also shows several of the most important settings using a nested <forms> tag. Namely, it sets the name of the security cookie, the length of time it will be considered valid (in minutes), and the page that allows the user to log in.

```
<configuration>
  ...
  <system.web>
    ...
      <authentication mode="Forms">
        <forms name="MyAppCookie"
          loginUrl="~/Login.aspx"
          protection="All"
          timeout="30" path="/" />
      </authentication>
    </system.web>
</configuration>
```

Note As usual, the ellipses (...) in the web.config code file listing represents the other settings that you be using in web.config file, which aren't related to security. These include the settings that Visual Studio automatically when you create the web.config file and any additional settings you've added to use specific features.

Table 19-1 describes these settings. They all supply default values, so you don't need to set them explicitly. For a complete list of supported attributes, consult the Visual Studio Help.

Table 19-1. Forms Authentication Settings

Attribute	Description
name	The name of the HTTP cookie to use for authentication (defaults to .ASPxAUTH). If multiple applications are running on the same web server, you should give each application's security cookie a unique name.
loginURL	Your custom login page, where the user is redirected if no valid authentication cookie is found. The default value is Login.aspx.
protection	The type of encryption and validation used for the security cookie (can be All, None, Encryption, or Validation). Validation ensures that the cookie isn't changed during transit, and encryption (typically Triple-DES) is used to encode its contents. The default value is All.
timeout	The number of idle minutes before the cookie expires. ASP.NET will refresh the cookie every time it receives a request. The default value is 30.
path	The path for cookies issued by the application. The default value (/) is recommended because case mismatches can prevent the cookie from being sent with a request.

Authorization Rules

If you make these changes to an application's web.config file and request a page, you'll notice that nothing unusual happens, and the web page is served in the normal way. This is because even though you have enabled forms authentication for your application, you have not restricted anonymous users. In other words, you've chosen the system you want to use for authentication, but at the moment none of your pages needs authentication.

To control who can and can't access your website, you need to add access-control rules to the <authorization> section of your web.config file. Here's an example that duplicates the default behavior:

```
<configuration>
  ...
  <system.web>
    ...
    <authentication mode="Forms">
      <forms loginUrl="~/Login.aspx" />
    </authentication>

    <authorization>
      <allow users="*" />
    </authorization>
  </system.web>
</configuration>
```

The asterisk (*) is a wildcard character that explicitly permits all users to use the application—even those who haven't been authenticated. But even if you don't include this line in your application's web.config file, this is still the behavior you'll see because ASP.NET's default settings allow all users. (Technically, this behavior

happens because there's an `<allow users = "*" />` rule in the root web.config file. If you're curious, you can find this file a directory like `c:\Windows\Microsoft.NET\Framework\[Version]\Config`, where [Version] is the version of ASP.NET that's installed, like v4.0.30319.)

To change this behavior, you need to explicitly add a more restrictive rule, as shown here:

```
<authorization>
  <deny users="?" />
</authorization>
```

The question mark (?) is a wildcard character that matches all anonymous users. By including this rule in your web.config file, you specify that anonymous users are not allowed. Every user must be authenticated, and every user request will require the security cookie. If you request a page in the application directory now, ASP.NET will detect that the request isn't authenticated. It will then redirect the request to the login page that's specified by the `login URL` attribute in the web.config file. (If you try this step right now, the redirection process will cause an error unless you've already created the login page.)

Now consider what happens if you add more than one rule to the authorization section:

```
<authorization>
  <allow users="*" />
  <deny users="?" />
</authorization>
```

When evaluating rules, ASP.NET scans the list from top to bottom and then continues with the settings in any .config file inherited from a parent directory, ending with the settings in the base machine.config file. As soon as it finds an applicable rule, it stops its search. Thus, in the previous case, it will determine that the rule `<allow users = "*" />` applies to the current request and will not evaluate the second line. This means that these rules will allow all users, including anonymous users.

But consider what happens if these two lines are reversed:

```
<authorization>
  <deny users="?" />
  <allow users="*" />
</authorization>
```

Now these rules will deny anonymous users (by matching the first rule) and allow all other users (by matching the second rule).

Controlling Access to Specific Directories

A common application design is to place files that require authentication in a separate directory. With ASP.NET configuration files, this approach is easy. Just leave the default `<authorization>` settings in the normal parent directory, and add a web.config file that specifies stricter settings in the secured directory. This web.config simply needs to deny anonymous users (all other settings and configuration sections can be omitted).

```
<!-- This web.config file is in a subfolder. -->
<configuration>
  <system.web>
    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

Note you cannot change the <authentication> tag settings in the web.config file of a subdirectory in your application. instead, all the directories in the application must use the same authentication system. However, each directory can have its own authorization rules.

Controlling Access to Specific Files

Generally, setting file access permissions by directory is the cleanest and easiest approach. However, you also have the option of restricting specific files by adding <location> tags to your web.config file.

The location tags sit outside the main <system.web> tag and are nested directly in the base <configuration> tag, as shown here:

```
<configuration>
  ...
  <system.web>
    ...
    <authentication mode="Forms">
      <forms loginUrl="~/Login.aspx" />
    </authentication>

    <authorization>
      <allow users="*" />
    </authorization>
  </system.web>

  <location path="SecuredPage.aspx">
    <system.web>
      <authorization>
        <deny users="?" />
      </authorization>
    </system.web>
  </location>

  <location path="AnotherSecuredPage.aspx">
    <system.web>
      <authorization>
        <deny users="?" />
      </authorization>
    </system.web>
  </location>
</configuration>
```

In this example, all files in the application are allowed except SecuredPage.aspx and AnotherSecuredPage.aspx, which have an additional access rule denying anonymous users. Notice that even when you use multiple <location> sections to supply different sets of authorization rules, you still only include one <authentication> section. That's because a web application can use only one type of authentication.

Tip you can also use the location tags to set rules for a specific subdirectory. it's up to you whether you to use this approach or prefer to create separate web.config files for each subdirectory, as described in the previous section.

Controlling Access for Specific Users

The <allow> and <deny> rules don't need to use the asterisk or question mark wildcards. Instead, they can specifically identify a user name or a list of comma-separated user names. For example, the following list specifically restricts access by three users. These users will not be able to access the pages in this directory. All other authenticated users will be allowed.

```
<authorization>
  <deny users="?" />
  <deny users="matthew,sarah" />
  <deny users="john" />
  <allow users="*" />
</authorization>
```

You'll notice that the first rule in this example denies all anonymous users. Otherwise, the following rules won't have any effect because users won't be forced to authenticate themselves.

The following rules explicitly allow two users. All other user requests will be denied access, even if they are authenticated.

```
<authorization>
  <deny users="?" />
  <allow users="matthew,sarah" />
  <deny users="*" />
</authorization>
```

Don't confuse these user names with the Windows user account names that are configured on your web server. When you use forms authentication, your application's security model is separate from the Windows user account system. Your application assigns the user name when a user logs in through the login page. Often, you'll choose user names that correspond to IDs in a database. The only requirement is that your user names be unique.

The WAT

You have another way to set up your authentication and authorization rules. Rather than edit the web.config file by hand, you can use the WAT from inside Visual Studio. The WAT guides you through the process, although you'll find it's still important to understand what changes are actually being made to your web.config file. It's also often quicker to enter a list of authorization rules by hand than to use the WAT.

To use the WAT for this type of configuration, select **Web Site** > **.NET Configuration** from the menu. Next, click the Security tab. You'll see the window shown in Figure 19-2, which gives you links to set the authentication type, define authorization rules (using the Access Rules section), and enable role-based security. (Role-based security is an optional higher-level feature you can use with forms authentication. You'll learn more about how it works and how to enable it in the next chapter.)

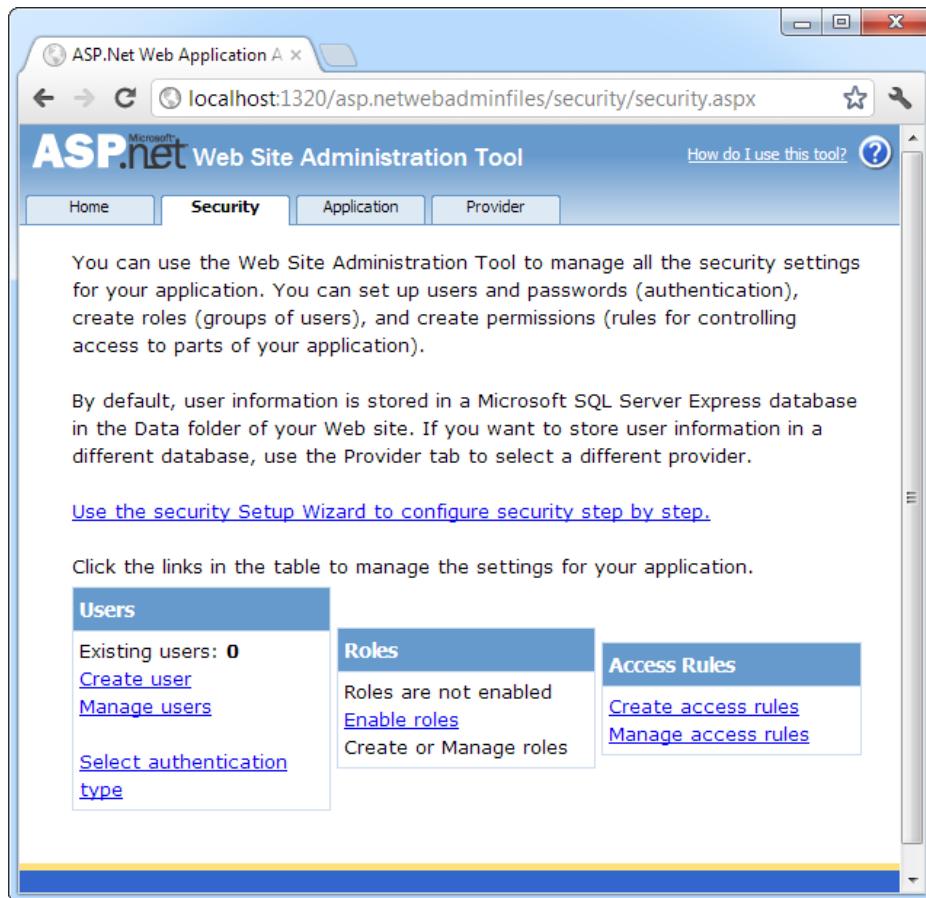


Figure 19-2. The Security tab in the WAT

To set an application to use forms authentication, follow these steps:

1. Click Select Authentication Type.
2. Choose the From the Internet option. (If you chose From a Local Network instead, you'll wind up using the built-in Windows authentication approach described later, in the "Windows Authentication" section.)
3. Click Done. The appropriate <authorization> tag will be created in the web.config file.

Tip The Select Authentication options are worded in a slightly misleading way. It's true that applications have users connecting from all over the internet are sure to use forms authentication. However, applications run on a local network might also use forms authentication—it all depends on how they connect and whether you want to use the information in existing accounts. In other words, a local intranet gives you the *option* to use authentication but doesn't require it.

Next, it's time to define the authorization rules. To do so, click the Create Access Rules link. (You can also change existing rules by clicking the Manage Access Rules link.) Using the slightly convoluted page shown in Figure 19-3, you have the ability to create a rule allowing or restricting specific users to the entire site or to specific pages or subfolders. For example, the rule in Figure 19-3 will deny the user jenny from the entire site (once you add it by clicking OK).

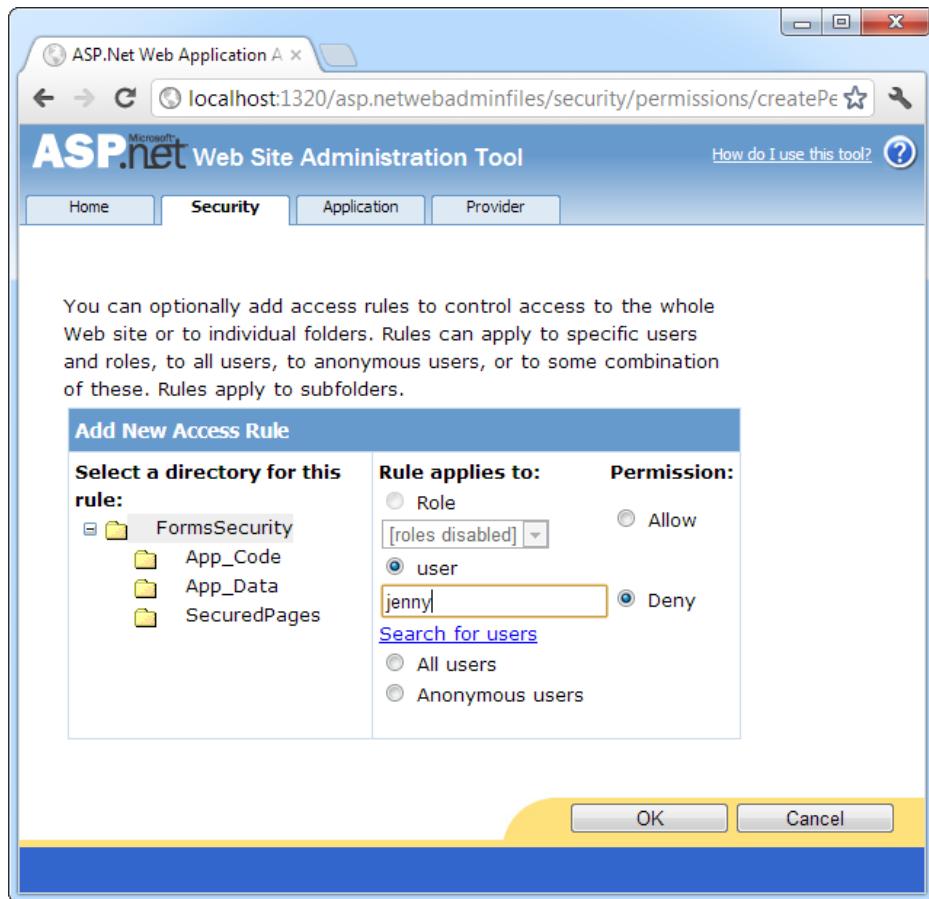


Figure 19-3. Adding an authorization rule

To manage multiple rules, you'll need to click the Manage Access Rules link. Now you'll have the chance to change the order of rules (and hence the priority, as described earlier), as shown in Figure 19-4. If you have a large number of rules to create, you may find it easier to edit the web.config file by hand. You might just want to create one initial rule to make sure it's in the right place and then copy and paste your way to success.

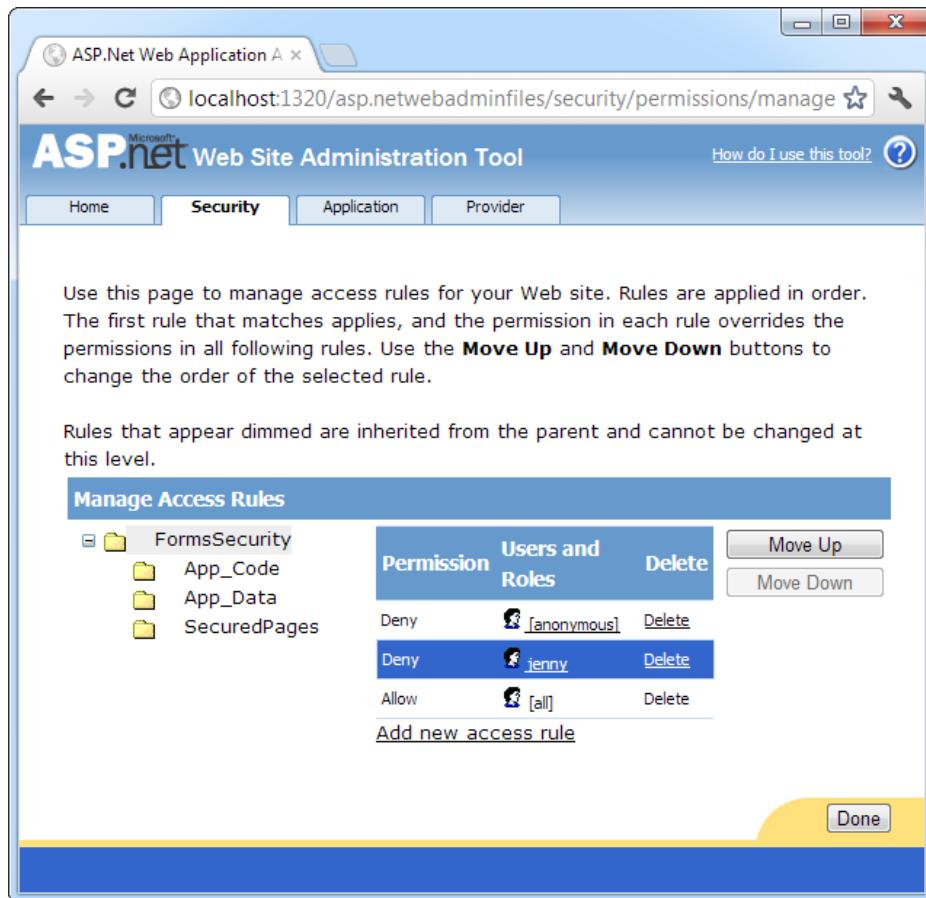


Figure 19-4. Ordering authorization rules

The Security tab is a little overwhelming at first glance because it includes a few features you haven't been introduced to yet. For example, the Security tab also allows you to create and manage user records and roles, as long as you're willing to use the prebuilt database structure that ASP.NET requires. You'll learn more about these details, which are part of a broad feature called *membership*, in the next chapter. For now, you'll concentrate on the authentication and authorization process.

The Login Page

Once you've specified the authentication mode and the authorization rules, you need to build the actual login page, which is an ordinary .aspx page that requests information from the user and decides whether the user should be authenticated.

ASP.NET provides a special `FormsAuthentication` class in the `System.Web.Security` namespace, which uses static methods that help manage the process. Table 19-2 describes the most important methods of this class.

Table 19-2. Members of the Forms Authentication Class

Member	Description
FormsCookieName	A read-only property that provides the name of the forms authentication cookie.
FormsCookiePath	A read-only property that provides the path set for the forms authentication cookie.
Authenticate()	Checks a user name and password against a list of accounts that can be entered in the web.config file.
RedirectFromLoginPage()	Logs the user into an ASP.NET application by creating the cookie, attaching it to the current response, and redirecting the user to the page originally requested.
SignOut()	Logs the user out of the ASP.NET application by removing the current encrypted cookie.
SetAuthCookie()	Logs the user into an ASP.NET application by creating and attaching the forms authentication cookie. Unlike the RedirectFromLoginPage() method, it doesn't take the user back to the initially requested page.
GetRedirectUrl()	Provides the URL of the originally requested page. You could use this with SetAuthCookie() to log a user into an application and make a decision in your code whether to redirect to the requested page or use a more suitable default page.
GetAuthCookie()	Creates the authentication cookie but doesn't attach it to the current response. You can perform additional cookie customization and then add it manually to the response, as described in Chapter 8.
HashPasswordForStoringInConfigFile()	Encrypts a string of text using the specified algorithm (SHA1 or MD5). This hashed value provides a secure way to store an encrypted password in a file or database.

A simple login page can put these methods to work with little code. To try it, begin by enabling forms authentication and denying anonymous users in the web.config, as described earlier:

```
<configuration>
  ...
  <system.web>
    ...
    <authentication mode="Forms">
      <forms loginUrl("~/Login.aspx" />
    </authentication>

    <authorization>
      <deny users="?" />
      <allow users="*" />
    </authorization>
  </system.web>
</configuration>
```

Now users will be redirected to a login page named Login.aspx that you need to create. Figure 19-5 shows an example of the simple login page that you might build.

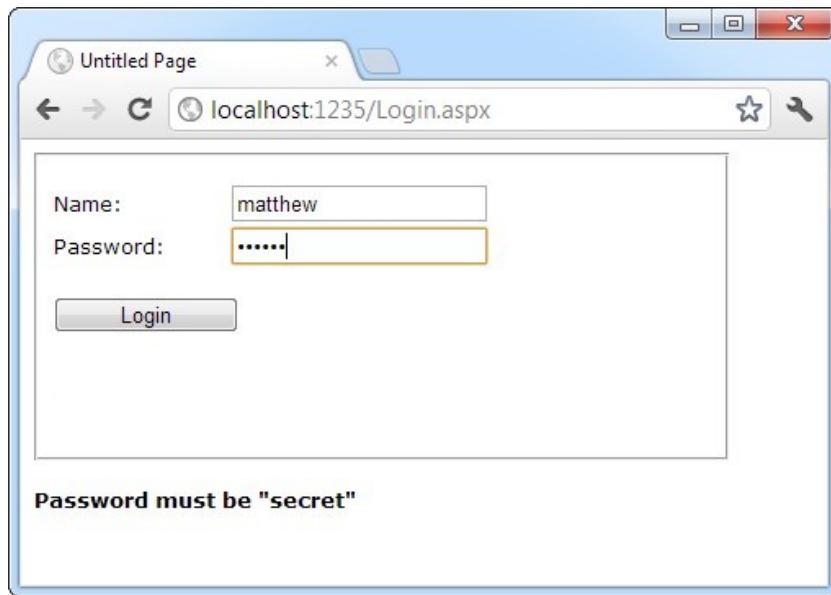


Figure 19-5. A typical login page

When the user clicks the Login button, the page checks whether the user has typed in the password *secret* and then uses the RedirectFromLoginPage() method to log the user in. Here's the complete page code:

```
public partial class Login : System.Web.UI.Page
{
    protected void cmdLogin_Click(Object sender, EventArgs e)
    {
        if (txtPassword.Text == "secret")
        {
            FormsAuthentication.RedirectFromLoginPage(
                txtName.Text, false);
        }
        else
        {
            lblStatus.Text = "Try again.";
        }
    }
}
```

The RedirectFromLoginPage() method requires two parameters. The first sets the name of the user. The second is a Boolean variable that specifies whether you want to create a persistent cookie (one that stays on the user's hard drive for a longer period of time). However, in an effort to be more secure, ASP.NET no longer honors this property in the way it was originally designed. (You'll learn more about persistent cookies shortly, in the section "Persistent Cookies.")

Obviously, the approach used in the login page isn't terribly secure—it simply checks that the user supplies a hard-coded password. In a real application, you'd probably check the user name and password against the information in a database and sign the user in only if the information matches exactly. You could write this code easily using the ADO.NET programming you learned in Part 4, although it requires a bit of tedious code. You'll consider more practical ways to accomplish this task in the next chapter.

You can test this code with the FormsSecurity website that's included with the downloadable code for this chapter. If you request the SecuredPage.aspx file, you'll be redirected to Login.aspx. After entering the correct password, you'll return to SecuredPage.aspx.

Retrieving the User's Identity

Once the user is logged in, you can retrieve the identity through the built-in User property, as shown here:

```
protected void Page_Load(Object sender, EventArgs e)
{
    lblMessage.Text = "You have reached the secured page, ";
    lblMessage.Text += User.Identity.Name + ".";
}
```

You don't need to place the code in the login page. Instead, you can use the User object to examine the current user's identity any time you need to do so.

Figure 19-6 shows the result of running this code.



Figure 19-6. Accessing a secured page

You can access the User object in your code because it's a property of the current Page object. The User object provides information about the currently logged-in user. It's fairly simple—in fact, User provides only one property and one method:

- The Identity property lets you retrieve the name of the logged-in user and the type of authentication that was used.
- The IsInRole() method lets you determine whether a user is a member of a given role (and thus should be given certain privileges). You'll use IsInRole() later in this chapter.

UnderSTANDING idenTiTiEs

The user object is standardized so that it can work with any type of authentication system. One consequence of this design is that the user.identity property returns a different type of object depending on the type of authentication you're using.

For example, when using forms authentication, the identity object is an instance of the FormsIdentity class. When using Windows authentication, you get a WindowsIdentity object instead. And if the user isn't logged in at all, you get a GenericIdentity. (All these classes implement the IIdentity interface, which standardizes them.)

most of the time, you don't need to worry about this sleight of hand. But occasionally you might want to cast the user.identity property to the more specific type to get access to an extra piece of information. For example, the FormsIdentity object provides the security ticket (in a property named Ticket), which isn't available through the standard IIdentity interface. This ticket is an instance of the FormsAuthenticationTicket class, and it provides a few miscellaneous details, like the time the user logged in and when the ticket will expire. Similarly, the WindowsIdentity object provides additional information that relates to Windows accounts (such as whether the current user is using a guest account or a system account). You'll see an example of this technique later in this chapter, in the "Windows Authentication" section.

SigNinG Out

Any web application that uses forms authentication should also feature a prominent logout button that destroys the forms authentication cookie:

```
private void cmdSignOut_Click(Object sender, EventArgs e)
{
    FormsAuthentication.SignOut();
    Response.Redirect("~/Login.aspx");
}
```

Tip In the next chapter, you'll learn how to simplify life with the login controls. These controls allow you to build login pages (and other types of security-related user interfaces) with no code. However, they require a feature—membership—in order to work.

Persistent Cookies

In some situations, you may be using the forms authentication login for personalization instead of security. In this situation, you may decide to allow *persistent cookies*. A persistent authentication cookie remains on the user's hard drive and keeps the user signed in for hours, days, or weeks—even if the user closes and reopens the browser.

Creating a persistent cookie requires a bit more code than creating a standard forms authentication cookie. Instead of using the RedirectFromLoginPage() method, you need to manually create the authentication ticket, set its expiration time, encrypt it, attach it to the request, and then redirect the user to the requested page. All of these tasks are easy, but it's important to perform them in the correct order. (When using nonpersistent cookies, the RedirectFromLoginPage() takes care of all these tasks automatically.)

Persistent cookies also present a potential security risk because another user could use the same computer to access the secured pages, without being forced to log in. If you want to allow the user to create a persistent cookie, you should make it optional because the user may want to access your site from a public or shared computer. Generally, sites that use this technique include a check box with text such as *Keep Me Logged In*.

The following code examines a check box named chkPersist. If it's selected, the code creates a persistent cookie that lasts for 20 days (although you can change this detail to whatever time interval you like).

```
// Perform the authentication.
if (txtPassword.Text == "secret")
{
    if (chkPersist.Checked)
    {
        // Calculate the total number of minutes in 20 days,
        // and use that for the timeout.
        int timeout = (int)TimeSpan.FromDays(20).TotalMinutes;

        // Create an authentication ticket.
        FormsAuthenticationTicket ticket = new
            FormsAuthenticationTicket(txtName.Text, true, cookietimeout);

        // Encrypt the ticket (so people can't steal it as it travels over
        // the Internet).
        string encryptedTicket = FormsAuthentication.Encrypt(ticket);

        // Create the cookie for the ticket, and put the ticket inside.
        HttpCookie cookie = new HttpCookie(FormsAuthentication.FormsCookieName,
            encryptedTicket);
        // Give the cookie and the authentication ticket the same expiration.
        cookie.Expires = ticket.Expiration;

        // Attach the cookie to the current response. It will now travel back to
        // the client, and then back to the web server with every new request.
        HttpContext.Current.Response.Cookies.Set(cookie);

        // Send the user to the originally requested page.
        string requestedPage = FormsAuthentication.GetRedirectUrl(txtName.text,
            false);
        Response.Redirect(requestedPage, true);
    }
    else
    {
        // Use the standard authentication method.
        FormsAuthentication.RedirectFromLoginPage(
            txtName.Text, false);
    }
}
```

It's worth noting that the `FormsAuthentication.SignOut()` method will always remove the forms authentication cookie, regardless of whether it is a normal cookie or a persistent cookie.

Windows Authentication

With Windows authentication, the web server takes care of the authentication process. ASP.NET simply makes this identity available to your code for your security checks.

When you use Windows authentication, you force users to log into IIS before they're allowed to access secure content in your website. The user login information can be transmitted in several ways (depending on the network environment, the requesting browser, and the way IIS is configured), but the end result is that the user is authenticated using a local Windows account. Typically, this makes Windows authentication best suited to intranet scenarios, in which a limited set of known users is already registered on a network server.

To implement Windows-based security with known users, you need to follow three steps:

1. Set the authentication mode to Windows authentication in the web.config file. (If you prefer a graphical tool, you can use the WAT during development or IIS Manager after deployment.)
2. Disable anonymous access for a directory by using an authorization rule.
3. Configure the Windows user accounts on your web server (if they aren't already present).

You'll walk through these steps in the following sections.

Note The built-in Visual Studio web server doesn't support anonymous users with Windows authentication instead, Visual Studio automatically logs you in to the test server using your Windows account. As a result, you need to use an authorization rule that denies anonymous users. However, it's still good practice to add it in case, at some point, you will deploy your application to IIS and need to explicitly deny anonymous users.

Web.config Settings

To use Windows authentication, you need to make sure the `<authentication>` element is set accordingly in your `web.config` file. Here's how:

```
<configuration>
  <system.web>
    ...
    <authentication mode="Windows" />

    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

At the moment, there's only one authorization rule, which uses the question mark to refuse all anonymous users. This step is critical for Windows authentication (as it is for forms authentication). Without this step, the user will never be forced to log in.

Ideally, you won't even see the login process take place. Instead, Internet Explorer will pass along the credentials of the current Windows user, which the web server uses automatically. The Visual Studio integrated web server always works this way. IIS also works this way, provided you've set up integrated Windows authentication (as described in Chapter 26) and the browser supports it.

You can also add `<allow>` and `<deny>` elements to specifically allow or restrict users from specific files or directories. Unlike with forms authentication, you need to specify the name of the server or domain where the account exists. For example, this rule allows the user account matthew, which is defined on the computer named WebServer:

```
<allow users = "WebServer\matthew" />
```

For a shortcut, you can use localhost (or just a period) to refer to an account on the current computer, as shown here:

```
<allow users = ".\matthew" />
```

You can also restrict certain types of users, provided their accounts are members of the same Windows group, by using the `roles` attribute:

```
<authorization>
  <deny users="?" />
  <allow roles=".\\SalesAdministrator,.\\SalesStaff" />
  <deny users=".\\matthew" />
</authorization>
```

In this example, all users who are members of the SalesAdministrator or SalesStaff groups will be automatically authorized to access ASP.NET pages in this directory. Requests from the user matthew will be denied unless he is a member of the SalesAdministrator or SalesStaff group. Remember, ASP.NET examines rules in the order they appear and stops when it finds a match. Reversing these two authorization lines would ensure that the user matthew was always denied, regardless of whether matthew belongs to the SalesAdministrator or SalesStaff group.

You can also examine a user's group membership programmatically in your code, as shown here. Since the string includes a backslash, you need to remember to double it, or you can turn off C# escaping with a preceding at sign (@).

```
protected void Page_Load(Object sender, EventArgs e)
{
    if (User.IsInRole(@"MyDomainName\SalesAdministrators"))
    {
        // Do nothing; the page should be accessed as normal because
        // the user has administrator privileges.
    }
    else
    {
        // Don't allow this page. Instead, redirect to the home page.
        Response.Redirect("Default.aspx");
    }
}
```

In this example, the code checks for membership in a custom Windows group called SalesAdministrators. If you want to check whether a user is a member of one of the built-in groups, you don't need to specify a computer or domain name. Instead, you use this syntax:

```
if (User.IsInRole(@"BUILTIN\Administrators"))
{
    // (Code goes here.)
}
```

For more information about the `<allow>` and `<deny>` rules and about configuring individual files and directories, refer to the discussion in the "Authorization Rules" section earlier in this chapter.

Note that you have no way to retrieve a list of available groups on the web server (that would violate security), but you can find out the names of the default built-in Windows roles using the System.Security.Principal.WindowsBuiltInRole enumeration. Table 19-3 describes these roles. Not all will apply to ASP.NET use, although Administrator, Guest, and User probably will.

Table 19-3. Default Windows Roles

Role	Description
AccountOperator	Users with the special responsibility of managing the User accounts on a computer or domain.
Administrator	Users with complete and unrestricted access to the computer or domain. (If the web server computer uses user account control [UAC], Windows will hold back administrator privileges from Administrator accounts to reduce the risk of viruses and other malicious code.)
BackupOperator	Users who can override certain security restrictions only as part of backing up or restore operations.
Guest	Like the User role but even more restrictive.
PowerUser	Similar to Administrator but with some restrictions.
PrintOperator	Like User but with additional privileges for taking control of a printer.
Replicator	Like User but with additional privileges to support file replication in a domain.
SystemOperator	Similar to Adminstrator with some restrictions. Generally, system operators manage a computer.
User	Users are prevented from making systemwide changes and can run only <i>certified applications</i> (see http://www.microsoft.com/windowsserver2008/en/us/isv.aspx for more information).

A Windows Authentication Test

One of the nice features of Windows authentication is that no login page is required. Depending on the authentication protocol you're using (see Chapter 26 for the complete details), the login process may take place automatically or the browser may show a login dialog box. Either way, you don't need to perform any additional work.

You can retrieve information about the currently logged-on user from the User object. As you learned earlier, the User object provides identity information through the User.Identity property. Depending on the type of authentication, a different identity object is used, and each Identity object can provide customized information. To get some additional information about the identity of the user who has logged in with Windows authentication, you can convert the generic IIdentity object to a WindowsIdentity object (which is defined in the System.Security.Principal namespace).

The following is a sample test page that uses Windows authentication (see Figure 19-7). To use this code as written, you need to import the System.Security.Principal namespace (where the WindowsIdentity class is defined).

```
public partial class SecuredPage : System.Web.UI.Page
{
    protected void Page_Load(Object sender, EventArgs e)
    {
        StringBuilder displayText = new System.Text.StringBuilder();
        displayText.Append("You have reached the secured page, ");
        displayText.Append(User.Identity.Name);

        WindowsIdentity winIdentity = (WindowsIdentity)User.Identity;
        displayText.Append("<br /><br />Authentication Type: ");
        displayText.Append(winIdentity.AuthenticationType);
        displayText.Append("<br />Anonymous: ");
        displayText.Append(winIdentity.IsAnonymous);
        displayText.Append("<br />Authenticated: ");
        displayText.Append(winIdentity.IsAuthenticated);
        displayText.Append("<br />Guest: ");
        displayText.Append(winIdentity.IsGuest);
        displayText.Append("<br />System: ");
        displayText.Append(winIdentity.IsSystem);
        displayText.Append("<br />Administrator: ");
        displayText.Append(User.IsInRole(@"BUILTIN\Administrators"));

        lblMessage.Text = displayText.ToString();
    }
}
```

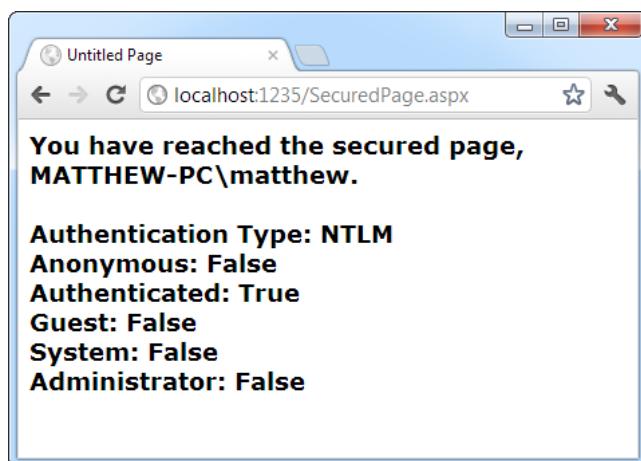


Figure 19-7. Retrieving Windows authentication information

The Last Word

In this chapter, you learned about the multilayered security architecture in ASP.NET and how you can safeguard your web pages and web services by using a custom login page or Windows authentication.

In the next chapter, you'll continue to build on your knowledge by considering some add-on features that can simplify your life and enhance your security. You'll learn how to get ASP.NET to create a basic user database for your site (complete with password encryption), saving you from creating it yourself or writing any ADO.NET code. You'll also extend your authorization rules by learning how you can group forms-authenticated users into logical groups, each of which can be assigned its own permissions. Altogether, these features are known as the ASP.NET *membership* system.



XML

XML is designed as an all-purpose format for organizing data. In many cases, when you decide to use XML, you're deciding to store data in a standardized way, rather than creating your own new (and to other developers, unfamiliar) format conventions. The actual location of this data—in memory, in a file, in a network stream—is irrelevant.

In this chapter, you'll learn the ground rules of the XML standard. You'll learn how to read XML content using the classes of the .NET library and how you can create and read your own XML documents. You'll also study some of the other standards that support and extend the basic rules of XML, including XML namespaces, XML schema, and XSLT.

XML Explained

The best way to understand the role XML plays is to consider the evolution of a simple file format *without* XML. For example, consider a simple program that stores product items as a list in a file. Say, when you first create this program you decide it will store three pieces of product information (ID, name, and price), and you'll use a simple text file format for easy debugging and testing. The file format you use looks like this:

```
1
Chair
49.33
2
Car
43399.55
3
Fresh Fruit Basket
49.99
```

This is the sort of format you might create using the .NET classes for writing files (such as the StreamWriter you learned about in Chapter 17). This format is easy to work with—you just write all the information, in order, from top to bottom. Of course, it's a fairly fragile format. If you decide to store an extra piece of information in the file (such as a flag that indicates whether an item is available), your old code won't work. Instead, you might need to resort to adding a header that indicates the version of the file:

SuperProProductList

Version 2.0

```
1
Chair
49.33
True
```

```

2
Car
43399.55
True
3
Fresh Fruit Basket
49.99
False

```

Now, you could check the file version when you open it and use different file-reading code appropriately. Unfortunately, as you add more and more possible versions, the file-reading code will become incredibly tangled, and you may accidentally break compatibility with one of the earlier file formats without realizing it. A better approach would be to create a file format that indicates where every product record starts and stops. Your code would then just set some appropriate defaults if it finds missing information in an older file format.

Here's a relatively crude solution that improves the SuperProProductList by adding a special sequence of characters (##Start##) to show where each new record begins:

SuperProProductList

Version 3.0

```
##Start##
```

```
1
```

```
Chair
```

```
49.33
```

```
True
```

```
##Start##
```

```
2
```

```
Car
```

```
43399.55
```

```
True
```

```
##Start##
```

```
3
```

```
Fresh Fruit Basket
```

```
49.99
```

```
False
```

All in all, this isn't a bad effort. Unfortunately, you may as well use the binary file format at this point—the text file is becoming hard to read, and it's even harder to guess what piece of information each value represents. On the code side, you'll also need some basic error-checking abilities of your own. For example, you should make your code able to skip over accidentally entered blank lines, detect a missing ##Start## tag, and so on, just to provide a basic level of protection.

The central problem with this homegrown solution is that you're reinventing the wheel. While you're trying to write basic file access code and create a reasonably flexible file format for a simple task, other programmers around the world are creating their own private, ad hoc solutions. Even if your program works fine and you can understand it, other programmers will definitely not find it easy.

Improving the List with XML

This is where XML comes into the picture. XML is an all-purpose way to identify any type of data using *elements*. These elements use the same sort of format found in an HTML file, but while HTML elements indicate formatting, XML elements indicate content. (Because an XML file is just about data, there is no standardized way to display it in a browser, although Internet Explorer shows a collapsible view that lets you show and hide different portions of the document.)

The SuperProProductList could use the following, clearer XML syntax:

```
<?xml version="1.0"?>
<SuperProProductList>
  <Product>
    <ID>1</ID>
    <Name>Chair</Name>
    <Price>49.33</Price>
    <Available>True</Available>
    <Status>3</Status>
  </Product>
  <Product>
    <ID>2</ID>
    <Name>Car</Name>
    <Price>43399.55</Price>
    <Available>True</Available>
    <Status>3</Status>
  </Product>
  <Product>
    <ID>3</ID>
    <Name>Fresh Fruit Basket</Name>
    <Price>49.99</Price>
    <Available>False</Available>
    <Status>4</Status>
  </Product>
</SuperProProductList>
```

This format is clearly understandable. Every product item is enclosed in a `<Product>` element, and every piece of information has its own element with an appropriate name. Elements are nested several layers deep to show relationships. Essentially, XML provides the basic element syntax, and you (the programmer) define the elements you want to use. That's why XML is often described as a *metalinguage*—it's a language you use to create your own language. In the SuperProProductList example, this custom XML language defines elements such as `<Product>`, `<ID>`, `<Name>`, and so on.

Best of all, when you read this XML document in most programming languages (including those in the .NET Framework), you can use XML parsers to make your life easier. In other words, you don't need to worry about detecting where an element starts and stops, collapsing whitespace, and so on (although you do need to worry about capitalization because XML is case sensitive). Instead, you can just read the file into some helpful XML data objects that make navigating the entire document much easier. Similarly, you can now extend the SuperProProductList with more information using additional elements, and any code you've already written will keep working without a hitch.

Note Although the examples in this chapter use XML to store data, XML is more commonly used to transfer data—for example, to let two applications running on different platforms communicate. One example is *services*—tiny code routines that you can call over the Internet. When an application asks a web service for data, the web service sends back the information in an XML document. That way, any program can read it, no matter what application framework was used to create it. You'll see a very simple web service example in Chapter 25.

XML FILEs VS. DataBaSeS

You can perform many tasks with XML—perhaps including some things it was never designed to do. This book is not intended to teach you XML programming but, rather, good ASP.NET application design. For ASP.NET programmers, XML file processing is an ideal replacement for custom file access routines and works best in situations where you need to store a small amount of data for relatively simple tasks.

XML files aren't a good substitute for a database because they have the same limitations as any other type of file access. In a web application, only a single user can update a file at a time without causing serious headaches, regardless of whether the file contains an XML document or binary content. Database products provide a far richer set of features for managing multiuser concurrency and providing optimal performance. Of course, nothing is stopping you from storing XML data *in* a database, which many database products actively encourage. In fact, the newest versions of leading database products such as SQL Server and Oracle even include extended XML features that support some of the standards you'll see in this chapter.

XML Basics

Part of XML's popularity is a result of its simplicity. When creating your own XML document, you need to remember only a few rules:

- XML elements are composed of a start tag (like <Name>) and an end tag (like </Name>). Content is placed between the start and end tags. If you include a start tag, you *must* also include a corresponding end tag. The only other option is to combine the two by creating an empty element, which includes a forward slash at the end and has no content (like <Name />). This is similar to the syntax for ASP.NET controls.
- Whitespace between elements is ignored. That means you can freely use tabs and hard returns to properly align your information.
- You can use only valid characters in the content for an element. You can't enter special characters, such as the angle brackets (<>) and the ampersand (&), as content. Instead, you have to use the entity equivalents (such as < and > for angle brackets, and & for the ampersand). These equivalents will be automatically converted to the original characters when you read them into your program with the appropriate .NET classes.
- XML elements are case sensitive, so <ID> and <id> are completely different elements.
- All elements must be nested in a root element. In the SuperProProductList example, the root element is <SuperProProductList>. As soon as the root element is closed, the document is finished, and you cannot add anything else after it. In other words, if you omit the <SuperProProductList> element and start with a <Product> element, you'll be able to enter information for only one product; this is because as soon as you add the closing </Product>, the document is complete. (HTML has a similar rule and requires that all page content be nested in a root <html> element, but most browsers let you get away without following this rule.)
- Every element must be fully enclosed. In other words, when you open a subelement, you need to close it before you can close the parent. <Product> <ID> </ID> </Product> is valid, but <Product> <ID> </Product> </ID> isn't. As a general rule, indent when you open a new element because this will allow you to see the document's structure and notice if you accidentally close the wrong element first.

- XML documents usually start with an XML declaration like <?xml version="1.0"?>. This signals that the document contains XML and indicates any special text encoding. However, many XML parsers work fine even if this detail is omitted.

As long as you meet these requirements, your XML document can be parsed and displayed as a basic tree. This means your document is well formed, but it doesn't mean it is valid. For example, you may still have your elements in the wrong order (for example, <ID> <Product> </Product> </ID>), or you may have the wrong type of data in a given field (for example, <ID> Chair</ID> <Name> 2</Name>). You can impose these additional rules on your XML documents, as you'll see later in this chapter when you consider XML schemas.

Elements are the primary units for organizing information in XML (as demonstrated with the SuperProProductList example), but they aren't the only option. You can also use *attributes*.

Attributes

Attributes add extra information to an element. Instead of putting information into a subelement, you can use an attribute. In the XML community, deciding whether to use subelements or attributes—and what information should go into an attribute—is a matter of great debate, with no clear consensus.

Here's the SuperProProductList example with ID and Name attributes instead of ID and Name subelements:

```
<?xml version="1.0"?>
<SuperProProductList>
  <Product ID="1" Name="Chair">
    <Price>49.33</Price>
    <Available>True</Available>
    <Status>3</Status>
  </Product>
  <Product ID="2" Name="Car">
    <Price>43399.55</Price>
    <Available>True</Available>
    <Status>3</Status>
  </Product>
  <Product ID="3" Name="Fresh Fruit Basket">
    <Price>49.99</Price>
    <Available>False</Available>
    <Status>4</Status>
  </Product>
</SuperProProductList>
```

Of course, you've already seen this sort of syntax with HTML elements and ASP.NET server controls:

```
<asp:DropDownList ID="lstBackColor" AutoPostBack="True"
  Width="194px" Height="22px" runat="server" />
```

Attributes are also common in the configuration file:

```
<sessionState mode="InProc" cookieless="false" timeout="20" />
```

XML has more stringent rules about attributes than HTML. In XML, attributes must always have values, and these values must use quotation marks. For example, <Product Name="Chair" /> is acceptable, but <Product Name=Chair /> or <Product Name /> isn't. However, you do have one bit of flexibility—you can use single or double quotes around any attribute value. It's convenient to use single quotes if you know the text value inside will contain a double quote (as in <Product Name='Red "Sizzle" Chair' />). If your text value has both single and double quotes, use double quotes around the value and replace the double quotes inside the value with the " entity equivalent.

Tip Order is not important when dealing with attributes. XML parsers treat attributes as a collection of unsorted information relating to an element. On the other hand, the order of elements often *is* important. To need a way of arranging information and preserving its order, or if you have a list of items with the same use elements, not attributes.

Comments

You can also add comments to an XML document. Comments go just about anywhere and are ignored for data processing purposes. Comments are bracketed by the <!-- and --> character sequences. The following listing includes three valid comments:

```
<?xml version="1.0"?>
<SuperProProductList>
    <!-- This is a test file. -->
    <Product ID="1" Name="Chair">
        <Price>49.33<!-- Why so expensive? --></Price>
        <Available>True</Available>
        <Status>3</Status>
    </Product>
    <!-- Other products omitted for clarity. -->
</SuperProProductList>
```

The only place you can't put a comment is embedded within a start or end tag (as in <myData <!-- A comment should not go here --> </myData>).

The XML Classes

.NET provides a rich set of classes for XML manipulation in several namespaces that start with System.Xml. One of the most confusing aspects of using XML with .NET is deciding which combination of classes you should use. Many of them provide similar functionality in a slightly different way, optimized for specific scenarios or for compatibility with specific standards.

The majority of the examples you'll explore use the types in the core System.Xml namespace. The classes here allow you to read and write XML files, manipulate XML data in memory, and even validate XML documents.

In this chapter, you'll look at the following options for dealing with XML data:

- Reading and writing XML directly, just as you read and write text files using XmlTextWriter and XmlTextReader. For sheer speed and efficiency, this is the best approach.
- Dealing with XML as a collection of in-memory objects using the XDocument class. If you need more flexibility than the XmlTextWriter and XmlTextReader provide, or you just want a simpler, more straightforward model (and you don't need to squeeze out every last drop of performance), this is a good choice.
- Using the Xml control to transform XML content to displayable HTML. In the right situation—when all you want to do is display XML content using a prebuilt XSLT style sheet—this approach offers a useful shortcut.

Note When it comes to XML, Microsoft is a bit schizophrenic. The .NET Framework includes at least a dozen ways to read and manipulate XML, including many that are too specialized or limited to cover in this chapter. In the following sections, you'll spend most of your time exploring the two most practical ways to work with XML. You'll learn to use the basic `XmlTextWriter` and `XmlTextReader` classes, which guarantee good performance. Finally, you'll explore the `XDocument` class, which can simplify intricate XML processing.

The XML TextWriter

One of the simplest ways to create or read any XML document is to use the basic `XmlTextWriter` and `XmlTextReader` classes. These classes work like their `StreamWriter` and `StreamReader` relatives, except that they write and read XML documents instead of ordinary text files. This means you follow the same process as you saw in Chapter 17 for creating a file. First, you create or open the file. Then, you write to it or read from it, moving from top to bottom. Finally, you close it and get to work using the retrieved data in whatever way you'd like.

Before beginning this example, you'll need to import the namespaces for file handling and XML processing:

```
using System.IO;  
using System.Xml;
```

Here's an example that creates a simple version of the `SuperProProductList` document:

```
// Place the file in the App_Data subfolder of the current website.  
// The System.IO.Path class makes it easy to build the full file name.  
string file = Path.Combine(Request.PhysicalApplicationPath,  
    @"App_Data\SuperProProductList.xml");  
  
FileStream fs = new FileStream(file, FileMode.Create);  
XmlTextWriter w = new XmlTextWriter(fs, null);  
  
w.WriteStartDocument();  
w.WriteStartElement("SuperProProductList");  
w.WriteComment("This file generated by the XmlTextWriter class.");  
  
// Write the first product.  
w.WriteStartElement("Product");  
w.WriteAttributeString("ID", "1");  
w.WriteAttributeString("Name", "Chair");  
  
w.WriteStartElement("Price");  
w.WriteString("49.33");  
w.WriteEndElement();  
  
w.WriteEndElement();  
  
// Write the second product.  
w.WriteStartElement("Product");  
w.WriteAttributeString("ID", "2");  
w.WriteAttributeString("Name", "Car");  
  
w.WriteStartElement("Price");  
w.WriteString("43399.55");
```

```
w.WriteLine();
w.WriteLine();

// Write the third product.
w.WriteStartElement("Product");
w.WriteAttributeString("ID", "3");
w.WriteAttributeString("Name", "Fresh Fruit Basket");

w.WriteStartElement("Price");
w.WriteString("49.99");
w.WriteEndElement();

w.WriteLine();

// Close the root element.
w.WriteLine();
w.WriteEndDocument();
w.Close();
```

Dissecting the Code . . .

- You create the entire XML document by calling the methods of the `XmlTextWriter`, in the right order. To start a document, you always begin by calling `WriteStartDocument()`. To end it, you call `WriteEndDocument()`.
- You write the elements you need, in three steps. First, you write the start tag (like `<Product>`) by calling `WriteStartElement()`. Then you write attributes, elements, and text content inside. Finally, you write the end tag (like `</Product>`) by calling `WriteEndElement()`.
- The methods you use always work with the current element. So if you call `WriteStartElement()` and follow it up with a call to `WriteAttributeString()`, you are adding an attribute to *that* element. Similarly, if you use `WriteString()`, you insert text content inside the current element, and if you use `WriteStartElement()` again, you write another element, nested inside the current element.

In some ways, this code is similar to the code you used to write a basic text file. It does have a few advantages, however. You can close elements quickly and accurately, the angle brackets (`<>`) are included for you automatically, and some errors (such as closing the root element too soon) are caught automatically, thereby ensuring a well-formed XML document as the final result.

When this code runs, it creates a file named `SuperProProductList.xml` in the `App_Data` folder of your website. If you want to see this file appear in the Solution Explorer, you need to refresh the display. To do that, right-click the `App_Data` folder and choose Refresh Folder.

To check that your code worked, open the file in Visual Studio by double-clicking it in the Solution Explorer (see Figure 18-1).

```

<?xml version="1.0"?>
<SuperProProductList>
    <!--This file generated by the XmlTextWriter class.-->
    <Product ID="1" Name="Chair">
        <Price>49.33</Price>
    </Product>
    <Product ID="2" Name="Car">
        <Price>43399.55</Price>
    </Product>
    <Product ID="3" Name="Fresh Fruit Basket">
        <Price>49.99</Price>
    </Product>
</SuperProProductList>

```

Figure 18-1. SuperProProductList.xml

ForMatting Your XML

By default, the `XmlTextWriter` will create an XML file that has all its elements lumped together in a single line without any helpful carriage returns or indentation. You don't see this limitation in Figure 18-1 because Internet Explorer uses a style sheet to give the XML a more readable (and more colorful) appearance. However, if you open the XML document in Notepad, you'll see the difference.

Although additional formatting isn't required (and doesn't change how the data will be processed), it makes a significant difference if you want to read your XML files in Visual Studio, Notepad, or another editor. Fortunately, the `XmlTextWriter` supports formatting; you just need to enable it, as follows:

```

// Set it to indent output.
w.Formatting = Formatting.Indented;

// Set the number of indent spaces.
w.Indentation = 5;

```

The XML Text Reader

Reading the XML document in your code is just as easy with the corresponding `XmlTextReader` class. The `XmlTextReader` moves through your document from top to bottom, one node at a time. You call the `Read()` method to move to the next node. This method returns true if there are more nodes to read or false once it has read the final node. The current node is provided through the properties of the `XmlTextReader` class, such as `NodeType` and `Name`.

A *node* is a designation that includes comments, whitespace, opening tags, closing tags, content, and even the XML declaration at the top of your file. To get a quick understanding of nodes, you can use the `XmlTextReader`

to run through your entire document from start to finish and display every node it encounters. The code for this task is as follows:

```
string file = Path.Combine(Request.PhysicalApplicationPath,
    @"App_Data\SuperProProductList.xml");
FileStream fs = new FileStream(file, FileMode.Open);
XmlTextReader r = new XmlTextReader(fs);

// Use a StringWriter to build up a string of HTML that
// describes the information read from the XML document.
StringWriter writer = new StringWriter();

// Parse the file, and read each node.
while (r.Read())
{
    // Skip whitespace.
    if (r.NodeType == XmlNodeType.Whitespace) continue;

    writer.Write("<b>Type:</b> ");
    writer.Write(r.NodeType.ToString());
    writer.Write("<br>");

    // The name is available when reading the opening and closing tags
    // for an element. It's not available when reading the inner content.
    if (r.Name != "")
    {
        writer.Write("<b>Name:</b> ");
        writer.Write(r.Name);
        writer.Write("<br>");
    }

    // The value is when reading the inner content.
    if (r.Value != "")
    {
        writer.Write("<b>Value:</b> ");
        writer.Write(r.Value);
        writer.Write("<br>");
    }

    if (r.AttributeCount > 0)
    {
        writer.Write("<b>Attributes:</b> ");
        for (int i = 0; i < r.AttributeCount; i++)
        {
            writer.Write(" ");
            writer.Write(r.GetAttribute(i));
            writer.Write(" ");
        }
        writer.Write("<br>");
    }
    writer.Write("<br>");
}
```

```
fs.Close();

// Copy the string content into a label to display it.
lblXml.Text = writer.ToString();
```

To test this, try the `XmlTextWriter.aspx` page included with the online samples. It produces the result shown in Figure 18-2.

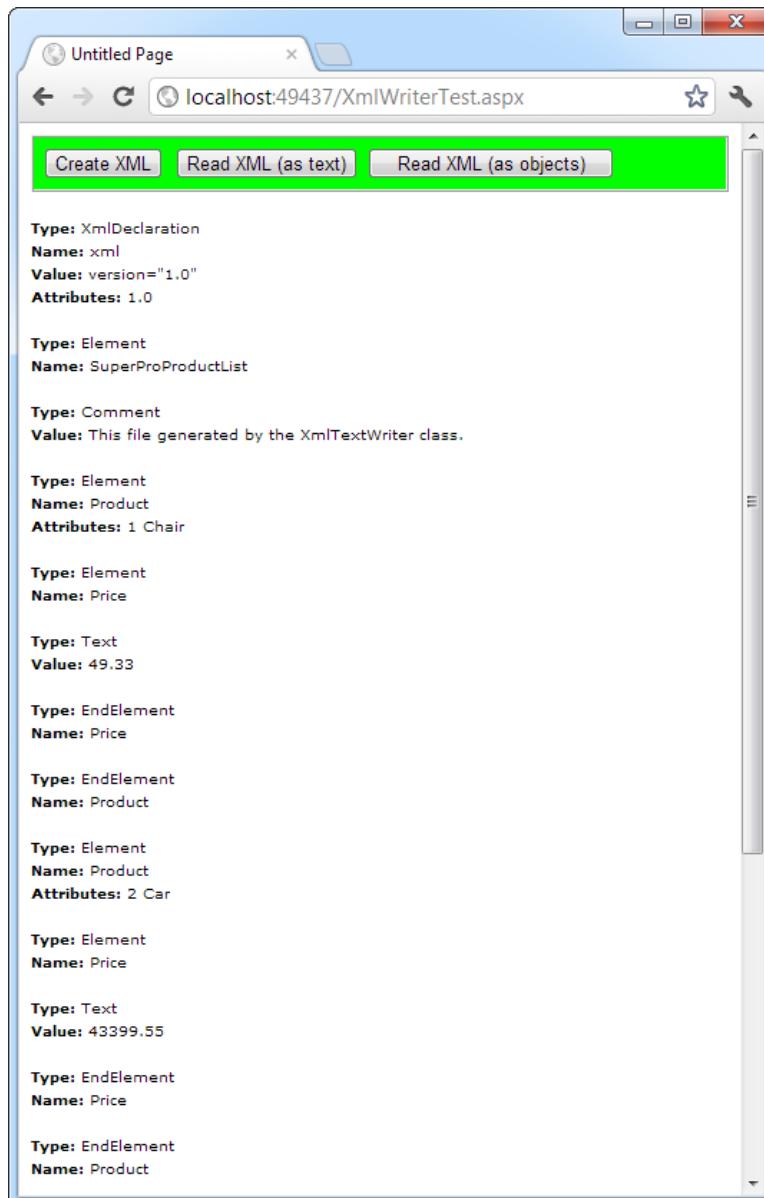


Figure 18-2. Reading XML structure

The following is a list of all the nodes that are found, shortened to include only one product:

Type: XmlDeclaration

Name: xml
Value: version="1.0"
Attributes: 1.0

Type: Element

Name: SuperProProductList

Type: Comment

Value: This file generated by the XmlTextWriter class.

Type: Element

Name: Product
Attributes: 1 Chair

Type: Element

Name: Price

Type: Text

Value: 49.33

Type: EndElement

Name: Price

Type: EndElement

Name: Product

Type: EndElement

Name: SuperProProductList

If you use the indentation trick described earlier (in the “Formatting Your XML” sidebar), you’ll see additional nodes that represent the bits of whitespace between elements.

In a typical application, you would need to go fishing for the elements that interest you. For example, you might read information from an XML file such as SuperProProductList.xml and use it to create Product objects based on the Product class shown here:

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

Nothing is particularly special about this class—all it does is allow you to store three related pieces of information (price, name, and ID). Note that this class uses automatic properties rather than public member variables, so its information can be displayed in a web page with ASP.NET data binding.

A typical application might read data from an XML file and place it directly into the corresponding objects. The next example (also a part of the XmlWriterTest.aspx page) shows how you can easily create a group of Product objects based on the SuperProProductList.xml file. This example uses the generic List collection, so you’ll need to import the System.Collections.Generic namespace.

```

// Open a stream to the file.
string file = Path.Combine(Request.PhysicalApplicationPath,
    @"App_Data\SuperProProductList.xml");
FileStream fs = new FileStream(file, FileMode.Open);
XmlTextReader r = new XmlTextReader(fs);

// Create a generic collection of products.
List<Product> products = new List<Product>();

// Loop through the products.
while (r.Read())
{
    if (r.NodeType == XmlNodeType.Element && r.Name == "Product")
    {
        Product newProduct = new Product();
        newProduct.ID = Int32.Parse(r.GetAttribute("ID"));
        newProduct.Name = r.GetAttribute("Name");

        // Get the rest of the subtags for this product.
        while (r.NodeType != XmlNodeType.EndElement)
        {
            r.Read();

            // Look for Price subtags.
            if (r.Name == "Price")
            {
                while (r.NodeType != XmlNodeType.EndElement)
                {
                    r.Read();
                    if (r.NodeType == XmlNodeType.Text)
                    {
                        newProduct.Price = Decimal.Parse(r.Value);
                    }
                }
            }
        }

        // You could check for other Product nodes
        // (such as Available, Status, etc.) here.
    }

    // Add the product to the list.
    products.Add(newProduct);
}

fs.Close();

// Display the retrieved document.
gridResults.DataSource = products;
gridResults.DataBind();

```

Dissecting the Code . . .

- This code uses a nested looping structure. The outside loop iterates over all the products, and the inner loop searches through all the child elements of <Product>. (In this example, the code processes the <Price> element and ignores everything else.) The looping structure keeps the code well organized.
- The EndElement node alerts you when a node is complete and the loop can end. Once all the information is read for a product, the corresponding object is added to the collection.
- All the information is retrieved from the XML file as a string. Thus, you need to use methods like Int32.Parse() to convert it to the right data type.
- Data binding is used to display the contents of the collection. A GridView set to generate columns automatically creates the table shown in Figure 18-3.

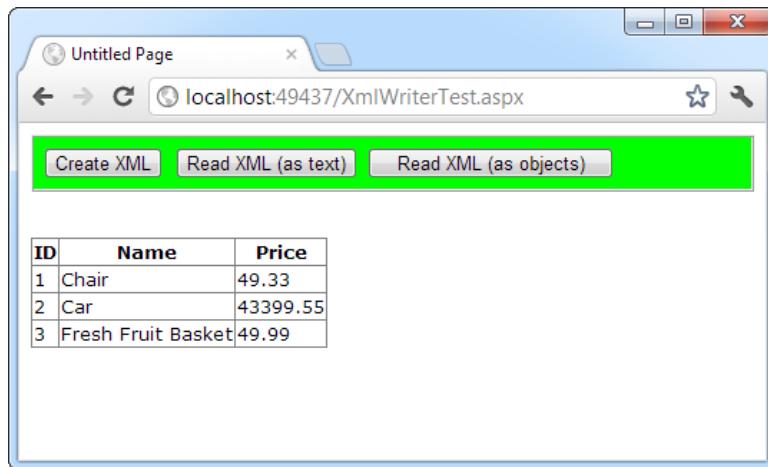


Figure 18-3. Reading XML content

Note The XmlTextReader provides many more properties and methods. These additional members do functionality; they allow for increased flexibility. For example, you can read a portion of an XML document string using methods such as ReadString(), ReadInnerXml(), and ReadOuterXml(). These members are all ed in the class library reference in the Visual Studio Help.

Working with XML Documents in Memory

The `XmlTextReader` and `XmlTextWriter` classes are streamlined for quickly getting XML data into and out of a file (or some other source). When using these classes, you open your XML file, retrieve the data you need, and use that data to create the appropriate objects or fill the appropriate controls. Your goal is to *translate* the XML into something more practical and usable. The rest of your code has no way of knowing that the data were initially extracted from an XML document—and it doesn't care.

Note Remember, the terms *XML document* and *XML file* are different. An XML document is a collection of elements structured according to the rules of XML. An XML document can be stored in virtually any way you want—it can be placed in a file, in a field, or in a database, or it can simply exist in memory. An XML file is a file that contains an XML document.

This approach is ideal for storing simple blocks of data. For example, you could modify the guest book page in the previous chapter to store guest book entries in an XML format, which would provide greater standardization but wouldn't change how the application works. Your code for serializing and deserializing the XML data would change, but the rest of the application would remain untouched.

The `XDocument` class provides a different approach to XML data. It provides an in-memory model of an entire XML document. You can then browse through the entire document, reading, inserting, or removing nodes at any location. (You can find the `XDocument` and all related classes in the `System.Xml.Linq` namespace.)

When using this approach, you begin by loading XML content from a file (or some other source) into an `XDocument` object. The `XDocument` holds the entire document at once, so it isn't a practical approach if your XML content is several megabytes in size. (If you have a huge XML document, the `XmlTextReader` and `XmlTextWriter` classes offer the best approach.) However, the `XDocument` really excels with the editing capabilities that it gives you. Using the `XDocument` object, you can manipulate the content or structure of any part of the XML document. When you're finished, you can save the content back to a file. Unlike the `XmlTextReader` and `XmlTextWriter`, the `XDocument` class doesn't maintain a direct connection to the file.

When you use the `XDocument` class, your XML document is created as a series of linked .NET objects in memory. Figure 18-4 shows the object model. (The diagram is slightly simplified from what you'll find when you start using the `XDocument` class—namely, it doesn't show the attributes, each of which is represented by an `XAttribute` object.)

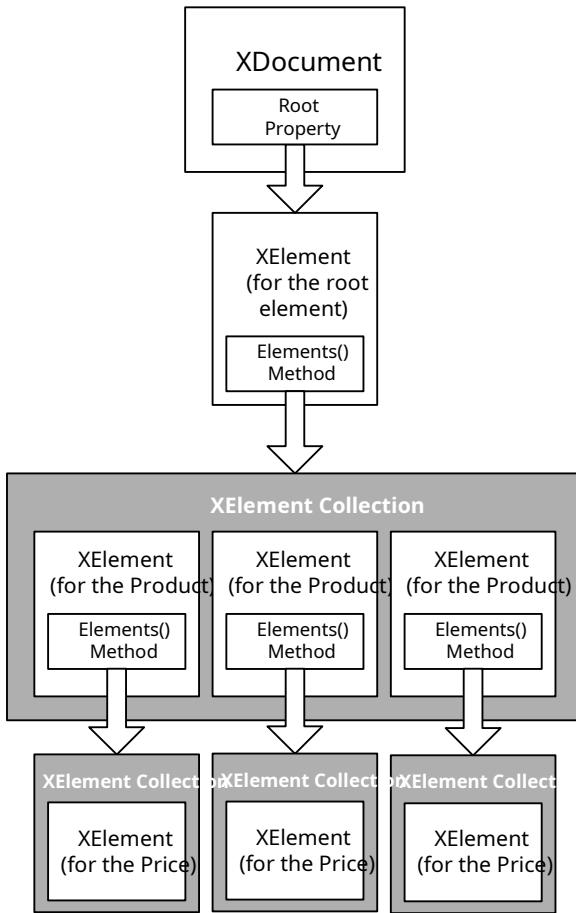


Figure 18-4. An XML document in memory

To start building a next XML document, you need to create the `XDocument`, `XElement`, and `XAttribute` objects that constitute it. All these classes have useful constructors that allow you to create and initialize them in one step. For example, you can create an element and supply text content that should be placed inside using code like this:

```
XElement element = new XElement("Price", 23.99);
```

This is already better than the `XmlTextWriter`, which forces you to start an element, insert its content, and close it with three separate statements. But the code savings become even more dramatic when you consider another feature of the `XDocument` and `XElement` classes—their ability to create a nested tree of nodes in a single code statement.

Here's how it works. Both the `XDocument` and `XElement` class include a constructor that takes a parameter array for the last argument. This parameter array holds a list of nested nodes.

Note A *parameter array* is a parameter that's preceded by the `params` keyword. This parameter is always the last parameter, and it's always an array. The advantage is that users don't need to declare the array; instead, you can simply tack on as many arguments as they want, which are grouped into a single array automatically.

Here's an example that creates an element with three nested elements and their content:

```
XElement element = new XElement("Product",
    new XElement("ID", 3),
    new XElement("Name", "Fresh Fruit Basket"),
    new XElement("Price", 49.99)
);
```

Here's the scrap of XML that this code creates:

```
<Product>
  <ID>3</ID>
  <Name>Fresh Fruit Basket</Name>
  <Price>49.99</Price>
</Product>
```

You can extend this technique to create an entire XML document, complete with elements, text content, attributes, and comments. For example, here's the complete code that creates the `SuperProProductList.xml` document in memory. When the document is completely built, the code saves it to a file using the `XDocument.Save()` method. (Although the file name isn't specified here, you can use the same code as shown earlier to set the file variable, placing it in the `App_Data` folder.)

```
// Build the XML content in memory.
XDocument doc = new XDocument(
    new XDeclaration("1.0", null, "yes"),
    new XComment("Created with the XDocument class."),
    new XElement("SuperProProductList",
        new XElement("Product",
            new XAttribute("ID", 1),
            new XAttribute("Name", "Chair"),
            new XElement("Price", 49.33)
        ),
        new XElement("Product",
            new XAttribute("ID", 2),
            new XAttribute("Name", "Car"),
            new XElement("Price", 43399.55)
        ),
        new XElement("Product",
            new XAttribute("ID", 3),
            new XAttribute("Name", "Fresh Fruit Basket"),
            new XElement("Price", 49.99)
        )
    )
);

// Save the document.
doc.Save(file);
```

This code creates the same XML content as the `XmlTextWriter` code you considered earlier. However, this code is shorter and easier to read.

Dissecting the Code . . .

- Every separate part of the XML document is created as an object. Elements are created as `XElement` objects, comments are created as `XComment` objects, and attributes are represented as `XAttribute` objects.
- Unlike the code that uses the `XmlTextWriter`, there's no need to explicitly close elements.
- Another nice detail is the way the indenting of the code statements mirrors the nesting of the elements in the XML document. If one element is followed by another and both elements have the same indenting, then the two elements are at the same level (for example, one `<Product>` element after another). If one element is followed by another and the second element has a greater indenting, it is being placed inside the preceding element (for example, the `<Price>` element in the `<Product>` element). The same holds true for other types of nodes, such as comments and attributes. This indenting allows you to look at your code and quickly take in the overall shape of the XML document.
- One of the best features of the `XDocument` class is that it doesn't rely on any underlying file. When you use the `Save()` method, the file is created, a stream is opened, the information is written, and the file is closed, all in one line of code. This means that this is probably the only line you need to put inside a try/catch error-handling block.

Figure 18-5 shows the file written by this code (as displayed by Visual Studio).

```

<?xml version="1.0"?>
<!--Created with the XDocument class.-->
<SuperProProductList>
  <Product ID="1" Name="Chair">
    <Price>49.33</Price>
  </Product>
  <Product ID="2" Name="Car">
    <Price>43399.55</Price>
  </Product>
  <Product ID="3" Name="Fresh Fruit Basket">
    <Price>49.99</Price>
  </Product>
</SuperProProductList>

```

Figure 18-5. The XML file

Reading an XML Document

The `XDocument` makes it easy to read and navigate XML content. You can use the static `XDocument.Load()` method to read XML documents from a file, URI, or stream, and you can use the static `XDocument.Parse()` method to load XML content from a string.

Once you have the XDocument with your content in memory, you can dig into the tree of nodes using a few key properties and methods of the XElement and XDocument class. Table 18-1 lists the most useful methods.

Table 18-1. Useful Methods for XElement and XDocument

Method	Description
Attributes()	Gets the collection of XAttribute objects for this element.
Attribute()	Gets the XAttribute with the specific name.
Elements()	Gets the collection of XElement objects that are contained by this element. (This is the top level only—these elements may in turn contain more elements.) Optionally, you can specify an element name, and only those elements will be retrieved.
Element()	Gets the single XElement contained by this element that has a specific name (or null if there's no match). If there is more than one matching element, this method gets just the first one.
Descendants()	Gets the collection of XElement objects that are contained by this element and (optionally) have the name you specify. Unlike the Elements() method, this method goes through all the layers of the document and finds elements at any level of the hierarchy.
Nodes()	Gets all the XNode objects contained by this element. This includes elements and other content, such as comments. However, unlike the XmlTextReader class, the XDocument does not consider attributes to be nodes.
DescendantNodes()	Gets all the XNode object contained by this element. This method is like Descendants() in that it drills down through all the layers of nested elements.

These methods give you added flexibility to filter out just the elements that interest you. For example, when using the Elements() method, you have two overloads to choose from. You can get all the child elements (in which case you would supply no parameters) or get just those child elements that have a specific element name (in which case you would specify the element name as a string). For example, here's how you would get the root <SuperProProductList> element from an XDocument that contains the complete SuperProProductList.xml:

```
// Use the Element() method, because there is just one matching element.
XElement superProProductListElement = doc.Element("SuperProProductList");
```

You can then use this element as a starting point to dig deeper into the document. For example, if you want to find the child <Product> elements in the <SuperProProductList>, you would add this code:

```
// Use the Elements() method, because there are several matching elements.
var productElements = superProProductListElement.Elements("Product");
```

Here, the code uses the var statement to simplify the code line. (Technically, the Elements() method returns an IEnumerable<XElement> collection. This design gives the XDocument more flexibility. It means the Elements() method can return any collection it wants, as long as the collection supports the IEnumerable<T> interface.)

You can now go through all the <Product> elements using a loop, or just grab a single one by index number, like this:

```
XElement productElement = productElements.ElementAt(0);
```

Getting the text inside an XElement is easy. You simply need to cast the element to the appropriate data type, as shown here:

```
XElement priceElement = productElement.Element("Price");
decimal price = (decimal)priceElement;
```

This works because the XElement class defines specialized conversion operators. When you cast an XElement to a decimal value, for example, the XElement automatically retrieves its inner value and attempts to convert that to a decimal.

Setting the text content inside an element is nearly as easy. You simply assign the new content to the Value property, as shown here:

```
decimal newValue = (decimal)priceElement * 2;
priceElement.Value = newValue.ToString();
```

You can use the same approach to read and set attributes with the XAttribute class.

Here's a straightforward code routine that mimics the XML processing code you saw earlier with the XmlTextReader. It scans through the elements that are available, creates a list of products, and displays that in a grid.

```
// Load the document.
XDocument doc = XDocument.Load(file);

// Loop through all the nodes, and create the list of Product objects.
List<Product> products = new List<Product>();

foreach (XElement element in doc.Element("SuperProProductList").Elements("Product"))
{
    Product newProduct = new Product();
    newProduct.ID = (int)element.Attribute("ID");
    newProduct.Name = (string)element.Attribute("Name");

    newProduct.Price = (decimal)element.Element("Price");

    products.Add(newProduct);
}

// Display the results.
gridResults.DataSource = products;
gridResults.DataBind();
```

The XElement class offers quite a few more members. For example, you'll find members for quickly stepping from one node to the next (FirstNode, LastNode, NextNode, PreviousNode, and Parent), properties for testing for the presence of children (HasElements), attributes (HasAttributes), content (IsEmpty), and methods for inserting, removing, and otherwise manipulating the XML tree of nodes. For example, use Add() to place a new child element inside the current element (after any existing content); use AddFirst() to place a new child element inside the current element (before any existing content); use AddAfterSelf() to insert an element at the same level just after the current element; use AddBeforeSelf() to insert an element at the same level just before the current element; and so on. You can also use Remove(), RemoveNodes(), ReplaceWith(), and ReplaceNodes() to remove or replace elements and other nodes.

The following example shows how you can add a new product to the XDocument:

```
// Create the element for the new product.
 XElement新产品 = new XElement ("Product",
    new XAttribute("ID", 4),
```

```

new XAttribute("Name", "Magic Lantern"),
new XElement("Price", "76.95")
};

// Add the element to the end of the current product list.
doc.Element("SuperProProductList").Add(newProduct);

```

Tip Whether you use the XDocument or the XmlTextReader class depends on a number of factors. Generally speaking, use XDocument when you want to deal directly with XML, rather than just using XML as a way to persist information. It also gives you the ability to modify the structure of an XML document, and it allows you to insert information in a more flexible way (not just from start to finish). On the other hand, the XmlTextReader is better suited for dealing with large XML files because it won't attempt to load the entire document into memory at once.

Searching an XML Document

One of the nicest features of the XDocument is its support of searching, which allows you to find nodes when you know they are there—somewhere—but you aren't sure how many matches exist or where the elements are.

To search an XDocument, all you need to do is use the Descendants() or DescendantNodes() method. Both methods allow you to search through the entire document tree in one step. For example, here's how you can use Descendants() on the entire SuperProProductList.xml document to get a list of prices:

```

XDocument doc = XDocument.Load(file);

// Find the matches.
var results = doc.Descendants("Price");

// Display the results.
lblXml.Text = "<b>Found " + results.Count().ToString() + " Matches </b>";
lblXml.Text += " for the Price tag: </b><br /><br />";
foreach ( XElement result in results)
{
    lblXml.Text += result.Value + "<br />";
}

```

Figure 18-6 shows the result.

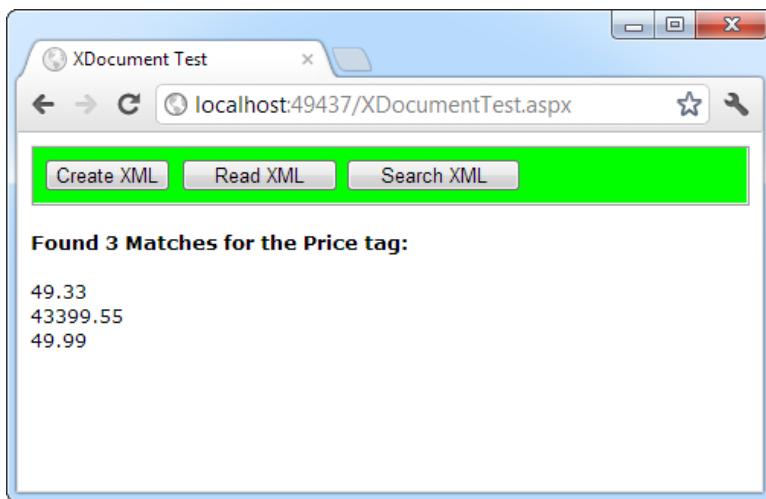


Figure 18-6. Searching an XML document

The `Descendants()` method is great if you want to find an element based on its name. If you want to use more sophisticated searching, match only part of a name, or examine only part of a document, you have two choices. First, you can write code that loops through all the nodes in the `XDocument` and checks each one. Second, you can use the LINQ to XML feature to perform a query that extracts matching `XElement` objects from your `XDocument`. This is a natural fit because the `XDocument` class was originally introduced as part of the LINQ feature in .NET 3.5. You'll learn much more about LINQ, including how to use it to perform searches, in Chapter 24.

XML Validation

XML has a rich set of supporting standards, many of which are far beyond the scope of this book. One of the most useful in this family of standards is XML Schema. XML Schema defines the rules to which a specific XML document should conform, such as the allowable elements and attributes, the order of elements, and the data type of each element. You define these requirements in an XML Schema document (XSD).

When you're creating an XML file on your own, you don't need to create a corresponding XSD file; instead, you might just rely on the ability of your code to behave properly. Although this is sufficient for tightly controlled environments, if you want to open your application to other programmers or allow it to interoperate with other applications, you should create an XSD. Think of it this way: XML allows you to create a custom language for storing data, and XSD allows you to define the syntax of the language you create.

XML Namespaces

Before you can create an XSD, you'll need to understand one other XML standard, called *XML namespaces*.

The core idea behind XML namespaces is that every XML markup language has its own namespace, which is used to uniquely identify all related elements. Technically, namespaces *disambiguate* elements by making it clear what markup language they belong to. For example, you could tell the difference between your SuperProProductList standard and another organization's product catalog because the two XML languages would use different namespaces.

Namespaces are particularly useful in compound documents, which contain separate sections, each with a different type of XML. In this scenario, namespaces ensure that an element in one namespace can't be confused with an element in another namespace, even if it has the same element name. Namespaces are also useful for applications that support different types of XML documents. By examining the namespace, your code can determine what type of XML document it's working with and can then process it accordingly.

Note XML namespaces aren't related to .NET namespaces. XML namespaces identify different XML languages. .NET namespaces are a code construct used to organize types.

Before you can place your XML elements in a namespace, you need to choose an identifying name for that namespace. Most XML namespaces use Universal Resource Identifiers (URIs). Typically, these URIs look like a web page URL. For example, <http://www.mycompany.com/mystandard> is a typical name for a namespace. Though the namespace looks like it points to a valid location on the Web, this isn't required (and shouldn't be assumed).

The reason that URIs are used for XML namespaces is that they are more likely to be unique. Typically, if you create a new XML markup, you'll use a URI that points to a domain or website you control. That way, you can be sure that no one else is likely to use that URI. For example, the namespace <http://www.SuperProProducts.com/SuperProProductList> is much more likely to be unique than just SuperProProductList if you own the domain www.SuperProProducts.com.

Tip Namespace names must match exactly. If you change the capitalization in part of a namespace, a trailing/character, or modify any other detail, it will be interpreted as a different namespace by the XML processor.

To specify that an element belongs to a specific namespace, you simply need to add the `xmlns` attribute to the start tag and indicate the namespace. For example, the `<Price>` element shown here is part of the <http://www.SuperProProducts.com/SuperProProductList> namespace:

```
<Price xmlns="http://www.SuperProProducts.com/SuperProProductList">
49.33
</Price>
```

If you don't take this step, the element will not be part of any namespace.

It would be cumbersome if you needed to type the full namespace URI every time you wrote an element in an XML document. Fortunately, when you assign a namespace in this fashion, it becomes the *default namespace* for all child elements. For example, in the XML document shown here, the `<SuperProProductList>` element and all the elements it contains are placed in the <http://www.SuperProProducts.com/SuperProProductList> namespace:

```
<?xml version="1.0"?>
<SuperProProductList
  xmlns="http://www.SuperProProducts.com/SuperProProductList">
  <Product>
    <ID>1</ID>
    <Name>Chair</Name>
    <Price>49.33</Price>
    <Available>True</Available>
    <Status>3</Status>
  </Product>
```

```
<!-- Other products omitted. -->
</SuperProProductList>
```

In compound documents, you'll have markup from more than one XML language, and you'll need to place different sections into different namespaces. In this situation, you can use namespace prefixes to sort out the different namespaces.

Namespace prefixes are short character sequences that you can insert in front of a tag name to indicate its namespace. You define the prefix in the xmlns attribute by inserting a colon (:) followed by the characters you want to use for the prefix. Here's the SuperProProductList document rewritten to use the prefix *super*:

```
<?xml version="1.0"?>
<super:SuperProProductList
  xmlns:super="http://www.SuperProProducts.com/SuperProProductList">
  <super:Product>
    <super:ID>1</super:ID>
    <super:Name>Chair</super:Name>
    <super:Price>49.33</super:Price>
    <super:Available>True</super:Available>
    <super:Status>3</super:Status>
  </super:Product>

  <!-- Other products omitted. -->
</super:SuperProProductList>
```

Namespace prefixes are simply used to map an element to a namespace. The actual prefix you use isn't important as long as it remains consistent throughout the document. By convention, the attributes that define XML namespace prefixes are usually added to the root element of an XML document.

Although the xmlns attribute looks like an ordinary XML attribute, it isn't. The XML parser interprets it as a namespace declaration. (The reason XML namespaces use XML attributes is a historical one. This design ensured that old XML parsers that didn't understand namespaces could still read newer XML documents that use them.)

Note Attributes act a little differently than elements when it comes to namespaces. You can use namespace prefixes with both elements and attributes; however, attributes don't pay any attention to the default namespace of a document. That means that if you don't add a namespace prefix to an attribute, the attribute will *not* belong to the default namespace. Instead, it will have no namespace.

Writing XML Content with Namespaces

You can use the XmlTextWriter and XDocument classes you've already learned to create XML content that uses a namespace.

The XmlTextWriter includes an overloaded version of the WriteStartElement() method that accepts a namespace URI. Here's how it works:

```
string ns = "http://www.SuperProProducts.com/SuperProProductList";
w.WriteStartDocument();
w.WriteStartElement("SuperProProductList", ns);

// Write the first product.
w.WriteStartElement("Product" , ns);
...
```

The only trick is to remember to use the namespace for every element.

The XDocument class deals with namespaces using a similar approach. First, you define an XNamespace object. Then you add this XNamespace object to the beginning of the element name every time you create an XElement (or an XAttribute) that you want to place in that namespace. Here's an example:

```
XNamespace ns = "http://www.SuperProProducts.com/SuperProProductList";
XDocument doc = new XDocument(
    new XDeclaration("1.0", null, "yes"),
    new XComment("Created with the XDocument class."),
    new XElement(ns + "SuperProProductList",
        new XElement(ns + "Product",
            new XAttribute("ID", 1),
            new XAttribute("Name", "Chair"),
            new XElement(ns + "Price", "49.33")
        ),
    ...
)
```

You may also need to change your XML reading code. If you're using the straightforward XmlTextReader, life is simple, and your code will work without any changes. If necessary, you can use the XmlTextReader. NamespaceURI property to get the namespace of the current element (which is important if you have a compound document that fuses elements from different namespaces).

If you're using the XDocument class, you need to take the XML namespace into account when you search the document. For example, when using the XElement.Element() method, you must supply the fully qualified element name by adding the appropriate XNamespace object to the string with the element name:

```
XNamespace ns = "http://www.SuperProProducts.com/SuperProProductList";
...
 XElement superProProductListElement = doc.Element(ns + "SuperProProductList");
```

Note Technically, you don't need to use the XNamespace class, although it makes your code clearer. When you add the XNamespace to an element name string, the namespace is simply wrapped in curly braces. In other words, when you combine the namespace `http://www.somecompany.com/DVDList` with the element name `Title`, it's equivalent to the string `{http://www.somecompany.com/DVDList}Title`. This syntax works because the curly brace characters aren't allowed in ordinary element names, so there's no possibility for confusion.

XML Schema Definition

An XSD, or *schema*, defines what elements and attributes a document should contain and the way these nodes are organized (the structure). It can also identify the appropriate data types for all the content. Schema documents are written using an XML syntax with specific element names. All the XSD elements are placed in the <http://www.w3.org/2001/XMLSchema> namespace. Often, this namespace uses the prefix `xsd:` or `xs:` (although it really doesn't matter).

The full XSD specification is out of the scope of this chapter, but you can learn a lot from a simple example. The following is a slightly abbreviated SuperProProductList.xsd file that defines the rules for SuperProProductList documents:

```
<?xml version="1.0"?>
<xsd:schema
    targetNamespace="http://www.SuperProProducts.com/SuperProProductList"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" >
```

```

<xs:element name="SuperProProductList">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="Product">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Price" Type="xs:double" />
          </xs:sequence>
            <xs:attribute name="ID" use="required" Type="xs:int" />
            <xs:attribute name="Name" use="required" Type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

At first glance, this markup looks a bit intimidating. However, it's actually not as complicated as it looks. Essentially, this schema indicates that a SuperProProductList document consists of a list of `<Product>` elements. Each `<Product>` element is a complex type made up of a string (Name), a decimal value (Price), and an integer (ID). This example uses the second version of the SuperProProductList document to demonstrate how to use attributes in a schema file.

Dissecting the Code . . .

- By examining the SuperProProductList.xsd schema, you can learn a few important points:
- Schema documents use their own form of XML markup. In the previous example, the elements are placed in the <http://www.w3.org/2001/XMLSchema> namespace using the `xs` namespace prefix.
- Every schema document starts with a root `<schema>` element.
- The schema document must specify the namespace of the documents it can validate. It specifies this detail with the `targetNamespace` attribute on the root `<schema>` element.
- The elements inside the `<schema>` element describe the structure of the target document. The `<element>` element represents an element, while the `<attribute>` element represents an attribute. To find out what the name of an element or attribute is, look at the `name` attribute. For example, you can tell quite easily that the first `<element>` has the name `SuperProProductList`. This indicates that the first element in the validated document must be `<SuperProProductList>`.
- If an element can contain other elements or has attributes, it's considered a *complex type*. Complex types are represented in a schema by the `<complexType>` element. The simplest complex type is a *sequence*, which is represented in a schema by the `<sequence>` element. It requires that elements are always in the same order—the order that's set in the schema document.
- When defining elements, you can define the maximum number of times an `element` appear (using the `maxOccurs` attribute) and the minimum number of times it *must* occur (using the `minOccurs` attribute). If you leave out these details, the default value of both is 1, which means that every element must appear exactly once in the target document. Use a `maxOccurs` value of *unbounded* if you want to allow an unlimited list. For example, this

allows an unlimited number of <Product> elements in the SuperProProductList catalog. However, the <Price> element must occur exactly once in each <Product>.

- When defining an attribute, you can use the `attribute` with a value of *required* to make that attribute mandatory.
- When defining elements and attributes, you can specify the data type using the `type` attribute. The XSD standard defines 44 data types that map closely to the basic data types in .NET, including the double, int, and string data types used in this example.

Validating an XML Document

The following example shows you how to validate an XML document against a schema, using an `XmlReader` that has validation features built in.

The first step when performing validation is to import the `System.Xml.Schema` namespace, which contains types such as `XmlSchema` and `XmlSchemaCollection`:

```
using System.Xml.Schema;
```

You must perform two steps to create the validating reader. First, you create an `XmlReaderSettings` object that specifically indicates you want to perform validation. You do this by setting the `ValidationType` property and loading your XSD schema file into the `Schemas` collection, as shown here:

```
// Configure the reader to use validation.
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;

// Create the path for the schema file.
string schemaFile = Path.Combine(Request.PhysicalApplicationPath,
    @"App_Data\SuperProProductList.xsd");

// Indicate that elements in the namespace
// http://www.SuperProProducts.com/SuperProProductList should be
// validated using the schema file.
settings.Schemas.Add("http://www.SuperProProducts.com/SuperProProductList",
    schemaFile);
```

Second, you need to create the validating reader using the static `XmlReader.Create()` method. This method has several overloads, but the version used here requires a `FileStream` (with the XML document) and the `XmlReaderSettings` object that has your validation settings:

```
// Open the XML file.
FileStream fs = new FileStream(file, FileMode.Open);

// Create the validating reader.
XmlReader r = XmlReader.Create(fs, settings);
```

The `XmlReader` in this example works in the same way as the `XmlTextReader` you've been using up until now, but it adds the ability to verify that the XML document follows the schema rules. This reader throws an exception (or raises an event) to indicate errors as you move through the XML file.

The following example shows how you can create a validating reader that uses the `SuperProProductList.xsd` file to verify that the XML in `SuperProProductList.xml` is valid:

```
// Set the validation settings.
XmlReaderSettings settings = new XmlReaderSettings();
```

```

settings.Schemas.Add("http://www.SuperProProducts.com/SuperProProductList",
    schemaFile);
settings.ValidationType = ValidationType.Schema;

// Open the XML file.
FileStream fs = new FileStream(file, FileMode.Open);

// Create the validating reader.
XmlReader r = XmlReader.Create(fs, settings);

// Read through the document.
while (r.Read())
{
    // Process document here.
    // If an error is found, an exception will be thrown.
}
fs.Close();

```

Using the current file, this code will succeed, and you'll be able to access each node in the document. However, consider what happens if you make the minor modification shown here:

```
<Product ID="A" Name="Chair">
```

Now when you try to validate the document, an `XmlSchemaException` (from the `System.Xml.Schema` namespace) will be thrown, alerting you to the invalid data type, as shown in Figure 18-7.

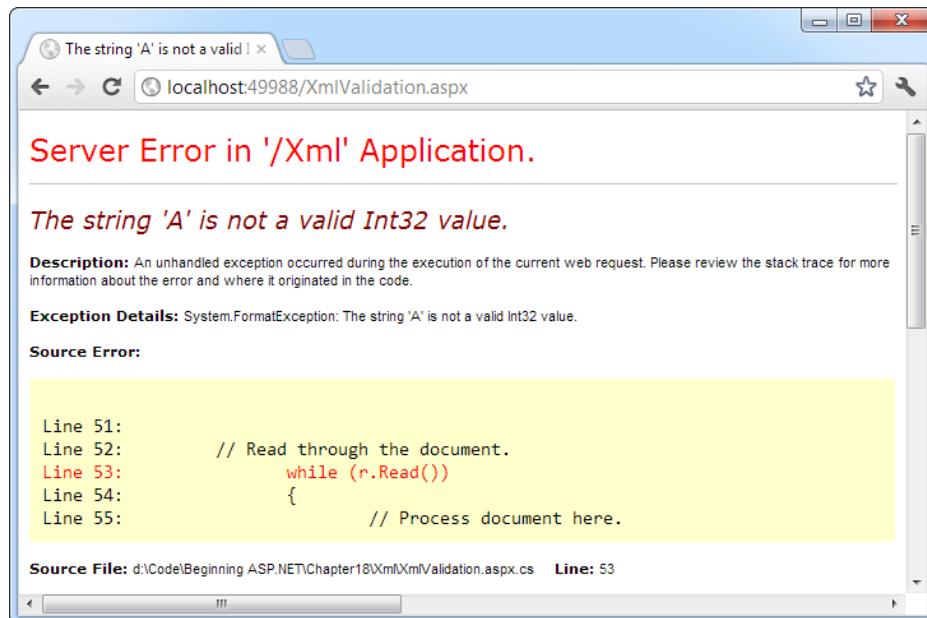


Figure 18-7. An `XmlSchemaException`

Instead of catching errors, you can react to the `XmlReaderSettings.ValidationEventHandler` event. If you react to this event, you'll be provided with information about the error, but no exception will be thrown. To connect an event handler to this event, you can attach an event handler before you create the `XmlReader`:

```
// Connect to the method named ValidateHandler.
settings.ValidationEventHandler += new ValidationEventHandler(ValidateHandler);
```

The event handler receives a `ValidationEventArgs` object as a parameter, which contains the exception, a message, and a number representing the severity:

```
public void ValidateHandler(Object sender, ValidationEventArgs e)
{
    lblStatus.Text += "Error: " + e.Message + "<br>";
}
```

To test the validation, you can use the `XmlValidation.aspx` page in the online samples. It allows you to validate a valid `SuperProProductList.xml` and two other versions, one with incorrect data and one with an incorrect element (see Figure 18-8).



Figure 18-8. The validation test page

Tip Because all `XmlReader` objects process XML one line at a time, this validation approach performs and uses the least amount of memory. But if you already have an `XDocument` in memory, you can validate similar way using the `XDocument.Validate()` method.

XML Display and Transforms

Another standard associated with XML is XSL Transformations (XSLT). XSLT allows you to create style sheets that can extract a portion of a large XML document or transform an XML document into another type of XML document. An even more popular use of XSLT is to convert an XML document into an HTML document that can be displayed in a browser.

Note eXtensible Stylesheet Language (XSL) is a family of standards for searching, formatting, and transforming XML documents. XSLT is the specific standard that deals with the transformation step.

XSLT is easy to use from the point of view of the .NET class library. All you need to understand is how to create an `XslCompiledTransform` object (found in the `System.Xml.Xsl` namespace). You use its `Load()` method to specify a style sheet and its `Transform()` method to output the result to a file or stream:

```
// Define the file paths this code uses. The XSLT file and the
// XML source file already exist, but the XML result file
// will be created by this code.
string xsitFile = Path.Combine(Request.PhysicalApplicationPath,
    @"App_Data\SuperProProductList.xsl");
string xmlSourceFile = Path.Combine(Request.PhysicalApplicationPath,
    @"App_Data\SuperProProductList.xml");
string xmlResultFile = Path.Combine(Request.PhysicalApplicationPath,
    @"App_Data\TransformedFile.xml");

// Load the XSLT style sheet.
XslCompiledTransform transformer = new XslCompiledTransform();
transformer.Load(xsitFile);

// Create a transformed XML file.
// SuperProProductList.xml is the starting point.
transformer.Transform(xmlSourceFile, xmlResultFile);
```

However, this doesn't spare you from needing to learn the XSLT syntax. Once again, the intricacies of XSLT aren't directly related to core ASP.NET programming, so they're outside the scope of this book. To get started with XSLT, however, it helps to review an example of a simple style sheet. The following shows an XSLT style sheet that transforms the no-namespace version of the `SuperProProductList` document into a formatted HTML table:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0" >

    <xsl:template match="SuperProProductList">
        <html>
            <body>
                <table border="1">
                    <xsl:apply-templates select="Product"/>
                </table>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="Product">
        <tr>
            <td><xsl:value-of select="@ID"/></td>
            <td><xsl:value-of select="@Name"/></td>
            <td><xsl:value-of select="Price"/></td>
        </tr>
    </xsl:template>
```

```
</xsl:template>
</xsl:stylesheet>
```

Every XSLT document has a root xsl:stylesheet element. The style sheet can contain one or more templates (the sample file SuperProProductList.xsl has two). In this example, the first template searches for the root SuperProProductList element. When it finds it, it outputs the tags necessary to start an HTML table and then uses the xsl:apply-templates command to branch off and perform processing for any contained Product elements.

```
<xsl:template match="SuperProProductList">
<html>
<body>
<table border="1">
<xsl:apply-templates select="Product"/>
```

When that process is complete, the HTML tags for the end of the table will be written:

```
</table>
</body>
</html>
</xsl:template>
```

When processing each <Product> element, the value from the nested ID attribute, Name attribute, and <Price> element is extracted and written to the output using the xsl:value-of command. The at sign (@) indicates that the value is being extracted from an attribute, not a subelement. Every piece of information is written inside a table row.

```
<xsl:template match="Product">
<tr>
<td><xsl:value-of select="@ID"/></td>
<td><xsl:value-of select="@Name"/></td>
<td><xsl:value-of select="Price"/></td>
</tr>
</xsl:template>
```

For more advanced formatting, you could use additional HTML elements to format some text in bold or italics.

The final result of this process is the HTML file shown here:

```
<html>
<body>
<table border="1">
<tr>
<td>1</td>
<td>Chair</td>
<td>49.33</td>
</tr>
<tr>
<td>2</td>
<td>Car</td>
<td>43398.55</td>
</tr>
<tr>
<td>3</td>
<td>Fresh Fruit Basket</td>
```

```
<td>49.99</td>
</tr>
</table>
</body>
</html>
```

In the next section, you'll look at how this output appears in an Internet browser.

Generally speaking, if you aren't sure you need XSLT, you probably don't. The .NET Framework provides a rich set of tools for searching and manipulating XML files using objects and code, which is the best approach for small-scale XML use.

Tip To learn more about XSLT, consider a book like Jeni Tennison's *Beginning XSLT 2.0: From Novice to Professional* (Apress, 2005).

The Xml Web Control

If you use an XSLT style sheet such as the one demonstrated in the previous example, you might wonder what your code should do with the generated HTML. You could try to write it directly to the browser or save it to the hard drive, but these approaches are awkward, especially if you want to display the generated HTML inside a normal ASP.NET web page that contains other controls. The `XslCompiledTransform` object just converts XML files—it doesn't provide any way to insert the output in your web page.

ASP.NET includes an `Xml` web control that fills the gap and can display XML content. You can specify the XML content for this control in several ways. For example, you can assign a string containing the XML content to the `DocumentContent` property, or you can specify a string that refers to an XML file using the `DocumentSource` property.

```
// Display the information from an XML file in the Xml control.
XmlProducts.DocumentSource = Path.Combine(Request.PhysicalApplicationPath,
    @"App_Data\SuperProProductList.xml");
```

If you assign the `SuperProProductList.xml` file to the `Xml` control, you're likely to be disappointed. The result is just a string of the inner text (the price for each product), bunched together without a space (see Figure 18-9).

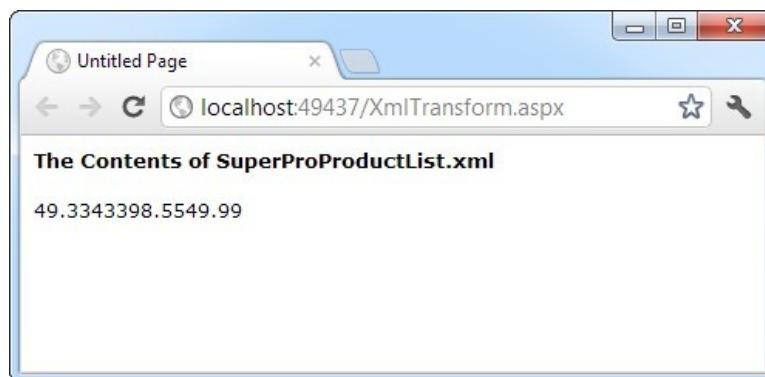


Figure 18-9. Unformatted XML content

However, you can also apply an XSLT style sheet, either by assigning an `XslCompiledTransform` object to the `Transform` property or by using a string that refers to the XSLT file with the `TransformSource` property:

```
// Specify a XSLT file.  
XmlProducts.TransformSource = Path.Combine(Request.PhysicalApplicationPath,  
    @"App_Data\SuperProProductList.xsl");
```

Now the output is automatically formatted according to your style sheet (see Figure 18-10).

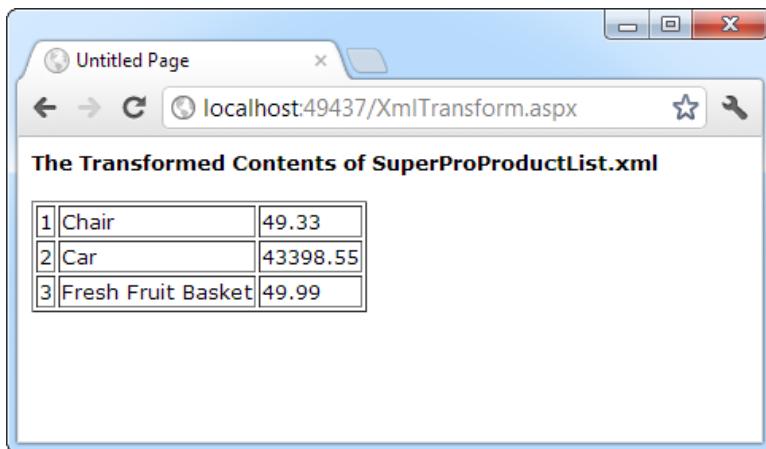


Figure 18-10. Transformed XML content

The Last Word

Now that your tour of XML and ASP.NET is drawing to a close, you should have a basic understanding of what XML is, how it looks, and why you might use it in a web page. XML is a valuable tool for breaking down the barriers between businesses and platforms—it's nothing less than a universal model for storing and communicating all types of information.

XML on its own is a remarkable innovation. However, to get the most of XML, you need to embrace other standards that allow you to validate XML, transform it, and search it for specific information. The .NET Framework provides classes for all these tasks in the namespaces under the `System.Xml` branch. To continue your exploration, start with a comprehensive review of XML standards (such as the one provided at <http://www.w3schools.com/xml>) and then dive into the class library.

Chapter 25



ASP.NET AJAX

So far, you've learned to build web pages that use the *postback* model. With this model, pages are perpetually being sent back to the web server and regenerated. For example, consider the greeting card maker you first saw in Chapter 6. When the user picks a font, enters some text, or chooses new colors, the page is posted back to the web server, allowing your code to run. The web page code makes the necessary adjustments, ASP.NET renders the page to HTML all over again, and the browser receives (and displays) the new version of the page.

This process seems somewhat labor intensive, but it's actually not that bad. The time required to complete the whole process (transmitting the posted-back page, running the code, rendering the page, and returning the final HTML) is surprisingly short. However, the process isn't seamless. When the greeting card is updated in the greeting card maker, the entire page is refreshed, including the parts that haven't changed. This produces a distracting flicker. This browser refresh is also a bit intrusive—for example, this process might interrupt users while they're in the middle of entering information in another control, or it might scroll them back to the beginning of the page even though they were previously looking at the end. The overall experience of using the greeting card maker is quite different from the experience of using a rich Windows application, which has no noticeable flicker and feels much more responsive.

Today there's a new generation of web applications that behave more like Windows applications than traditional web pages. These applications refresh themselves quickly and flicker-free and sometimes include slick new features such as animation and drag-and-drop. Notable examples include web-based e-mail applications such as Gmail and mapping tools such as Google Maps. Other websites—from Facebook to Flickr—use rich features more subtly to improve responsiveness and add frills.

This new breed of web applications uses a set of design practices and technologies known as Ajax. Ajax is programming shorthand for a set of techniques that create more-responsive, dynamic pages. One of the hallmarks of Ajax is the ability to refresh part of the page while leaving the rest untouched.

In this chapter, you'll learn how Ajax works and how you can use it to create rich, responsive web pages. You won't delve into the intricate details of do-it-yourself Ajax (which requires an extensive understanding of JavaScript), but you will explore the Ajax features of ASP.NET. These features allow you to use the familiar ASP.NET model—.NET objects and server-side controls—to create Ajax effects with surprisingly little effort.

Understanding Ajax

Before you really get started with Ajax, it's important to understand its capabilities and limitations. Only then will you know how to fit it into your web applications.

Ajax: The Good

The key benefit of Ajax is responsiveness. An Ajax application, when done properly, provides a better experience for the user. Even if the user can't do anything new (or do anything faster), this improved experience can make

your web application seem more modern and sophisticated. If you're creating a website that's competing against other similar sites, you just might find that Ajax allows you to distinguish your work from the rest of the pack.

Ajax can also provide genuinely new features that aren't possible in traditional web pages. For example, Ajax pages often use JavaScript code that reacts to client-side events such as mouse movements and key presses. These events occur frequently, so it's not practical to deal with them by using the postback model. For example, imagine you want to highlight a TextBox when the user moves the mouse over it. With the postback approach, you'd need to send the entire page back to the web server, regenerate it, and refresh it in the browser—by which point the mouse might be somewhere completely different. This approach is clearly impractical. However, an Ajax page can deal with this scenario because it can react immediately, updating the page if needed or requesting additional information from the web server in the background. While this request is under way, the user is free to keep working with the page. In fact, the user won't even realize that the request is taking place.

Note Ajax isn't really a whole new technology. More accurately, it's a set of techniques, some of which are existing practices. For example, you've already seen quite a few ASP.NET controls that use client-side JavaScript to provide a richer experience, such as the validation controls (Chapter 9) and the Menu control (Chapter 13). Ajax pages use much more JavaScript than normal, they often require interactions between controls, and they request additional information from the web server by using a special browser object called XMLHttpRequest, which is available to client-side JavaScript code.

Ajax: The Bad

There are two key challenges to using Ajax. The first is complexity. Writing the JavaScript code needed to implement an Ajax application is a major feat. Fortunately, you'll sidestep this problem in this chapter, because you'll use ASP.NET's Ajax-enabled features. That means you'll let Microsoft manage the complexity instead of worrying about it yourself.

The other challenge to using Ajax is browser support. In the past, this was a significant concern, but today it's rare to find a browser that doesn't support the necessary JavaScript features, even on mobile devices. ASP.NET is clever enough to use fallbacks when it encounters a browser that doesn't support Ajax (or has JavaScript switched off). For example, if you're using the partial rendering support that's provided by ASP.NET's UpdatePanel control (which you'll learn about later in this chapter), your page will simply use full postbacks instead of more-streamlined partial updates. However, this support has its limits, and if you're using a more advanced Ajax-enabled web control, you may find that it doesn't work properly on severely limited browsers. The only way to know is to experiment.

Finally, Ajax applications introduce a few quirks that might not be to your liking. Web pages that use Ajax often do a lot of work on a single page. This is different from traditional web pages, which often move the user from one page to another to complete a task. Although the multiple-page approach is a little more roundabout, it allows the user to place bookmarks along the way and use the browser's Back and Forward buttons to step through the sequence. These techniques usually don't work with Ajax applications, because there's only a single page to bookmark or navigate to, and the URL for that page doesn't capture the user's current state. This isn't a showstopper of an issue, but it might cause you to consider the design of your web application a little more closely.

Ajax in ASP.NET

There are a variety of ways to implement Ajax in any web application, including ASP.NET. To implement it on your own, you need to have a thorough understanding of JavaScript, because it's JavaScript code that runs in the browser, requesting the new information from the web server when needed and updating the page accordingly. Although JavaScript isn't terribly complex, it's remarkably difficult to program correctly, for two reasons:

- The implementation of key JavaScript details varies from browser to browser, which means you need a tremendous amount of experience to write a solid web page that runs equally well on all browsers (or the help of a higher-level JavaScript library, such as jQuery).
- JavaScript is a notoriously loose language that tolerates many minor typos and mistakes. Catching these mistakes and removing them is a tedious process. Even worse, the mistakes might be fatal in some browsers and harmless in others, which complicates debugging. (However, if you find yourself in the unenviable position of trying to correct misbehaving JavaScript code, you can get some help from Visual Studio's top-notch JavaScript debugger.)

In this chapter, you won't use JavaScript directly. Instead, you'll use a higher-level model called ASP.NET AJAX. ASP.NET AJAX gives you a set of server-side components and controls that you can use when designing your web page. These components automatically render all the JavaScript you need to get the effect you want. The result is that you can create a page with Ajax effects while programming with a familiar (and much more productive) model of server-side objects. Of course, you won't get quite as much control to customize every last detail about the way your web pages work, but you will get some great functionality with minimal effort.

Note It's generally accepted that Ajax isn't written in all capitals, because the word isn't an acronym. (Technically, it's a short form for *Asynchronous JavaScript and XML*, although this technique is now considered to be one of several possible characteristics of an Ajax web application.) However, Microsoft chose to write the word in uppercase when it named ASP.NET AJAX. Therefore, you'll see two capitalizations of Ajax in this chapter—Ajax talking in general terms about the technology and philosophy of Ajax, and *AJAX* when talking about ASP.NET, which is Microsoft's specific implementation of these concepts.

The ScriptManager

To use ASP.NET AJAX, you need to place a new web control on your page. This control is the ScriptManager, and it's the brains of ASP.NET AJAX.

Like all ASP.NET AJAX controls, the ScriptManager is placed on a Toolbox tab named **AJAX Extensions**. When you drag the ScriptManager onto your page, you'll end up with this declaration:

```
<asp:ScriptManager ID = "ScriptManager1" runat = "server" ></asp:ScriptManager>
```

At design time, the ScriptManager appears as a blank gray box. But when you request a page that uses the ScriptManager, you won't see anything, because the ScriptManager doesn't generate any HTML tags. Instead, the ScriptManager performs a different task—it adds the links to the ASP.NET AJAX JavaScript libraries. It does that by inserting a script block that looks something like this:

```
<script src = "/ScriptResource.axd?d = RUSU1mI ..." type = "text/javascript">
</script>
```

This script block doesn't contain any code. Instead, it uses the src attribute to pull the JavaScript code out of a separate file.

However, the ScriptManager is a bit craftier than you might expect. Rather than use a separate file to get its JavaScript (which would then need to be deployed along with your application), the src attribute uses a long, strange-looking URL that points to ScriptResource.axd. ScriptResource.axd isn't an actual file—instead, it's a resource that tells ASP.NET to find a JavaScript file that's embedded in one of the compiled .NET assemblies. The long query string argument at the end of the URL tells the ScriptResource.axd extension which file to send to the browser.

The JavaScript files that ASP.NET AJAX uses contain hundreds of lines of highly complex, concise code that forms the basis for all the Ajax features you'll see in this chapter. However, these files are quite compact, requiring the client to download less than 200KB of script code (depending on the features that you're using). When you're visiting an ASP.NET AJAX-powered site, the script code is downloaded only once, and then cached by the browser so it can be used in various ways by various pages in the website. (In addition, ASP.NET sends a compressed version of the script document, if the browser supports it. Currently, ASP.NET uses compression when receiving requests from Internet Explorer 7 or later.) The bottom line is that pages using ASP.NET AJAX features don't require significantly longer download times.

Each page that uses ASP.NET AJAX features requires an instance of the ScriptManager. However, you can use only one ScriptManager on a page. ASP.NET AJAX-enabled controls can interact with the ScriptManager, asking it to render links to additional JavaScript resources.

Tip If you're using ASP.NET AJAX features throughout your website, you might choose to place the ScriptManager in a master page. However, this can occasionally cause problems, because different content pages may want to set the properties of the ScriptManager differently. In this scenario, the solution is to use the ScriptManager on the master page and the ScriptManagerProxy in your content page. (You can find the ScriptManagerProxy on the same Extensions tab of the Toolbox.) Each content page can configure the ScriptManagerProxy control in the same way you would configure the ScriptManager—in fact, all the ScriptManagerProxy settings are applied to the ScriptManager control when the page is executed.

Now that you've taken a brief overview of Ajax, it's time to start building Ajax-enabled pages. In the next part of this chapter, you'll consider the following topics:

- Using partial refreshes to avoid full-page postbacks and page flicker
- Using progress notifications to deal with slower updates
- Using timed refreshes to automatically update a portion of your page
- Using the ASP.NET AJAX Control Toolkit to get a range of slick new controls with Ajax features

Using Partial Refreshes

The key technique in an Ajax web application is the *partial refresh*. With partial refreshes, the entire page doesn't need to be posted back and refreshed in the browser. Instead, when something happens, the web page asks the web server for more information. The request takes place in the background, so the web page remains responsive. (It's up to you whether to use some sort of progress indicator if you think the request might take a noticeable amount of time.) When the web page receives the response, it updates just the changed portion of the page, as shown in Figure 25-1.

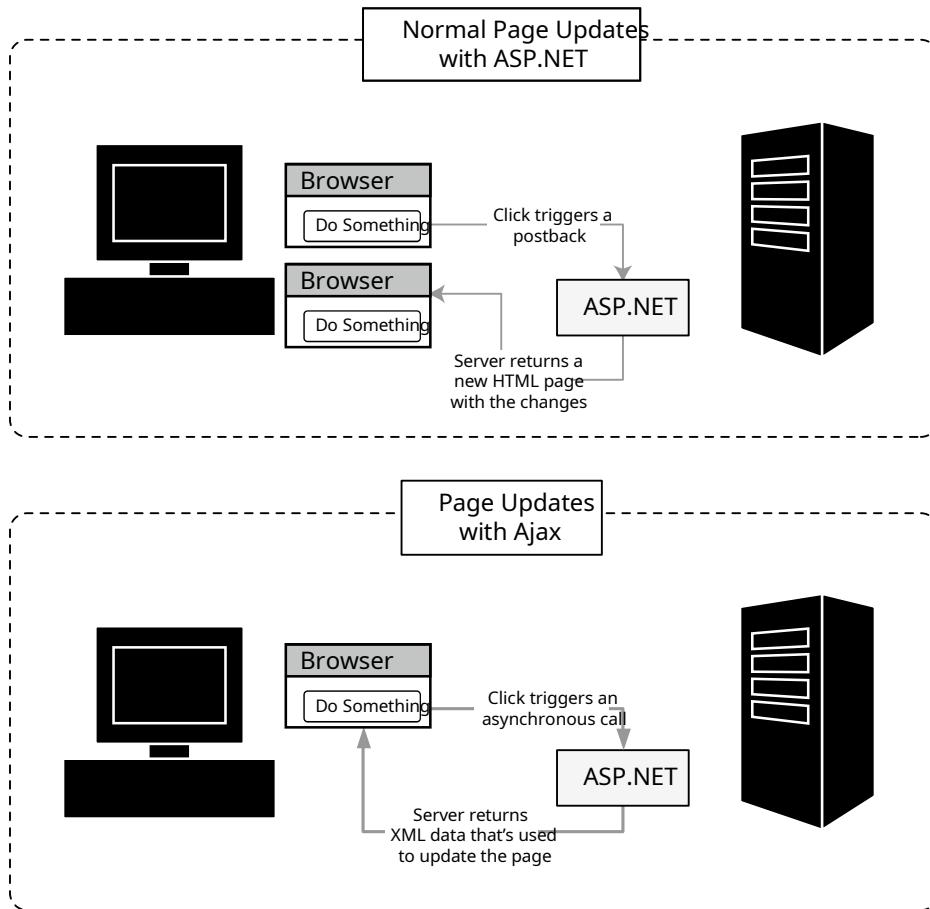


Figure 25-1. Ordinary server-side pages versus Ajax

ASP.NET includes a handy control that lets you take an ordinary page with server-side logic and make sure it refreshes itself in flicker-free Ajax style by using partial updates. This control is the UpdatePanel.

The basic idea is that you divide your web page into one or more distinct regions, each of which is wrapped inside an invisible UpdatePanel. When an event occurs in a control that's located inside an UpdatePanel, and this event would normally trigger a full-page postback, the UpdatePanel intercepts the event and performs an asynchronous callback instead. Here's an example of how it happens:

1. The user clicks a button inside an UpdatePanel.
2. The UpdatePanel intercepts the client-side click. ASP.NET AJAX performs a callback to the server instead of a full-page postback.
3. On the server, your normal page life cycle executes, with all the usual events. Finally, the page is rendered to HTML and returned to the browser.
4. ASP.NET AJAX receives HTML content for every UpdatePanel on the page. The client-side script code then updates the page, replacing the existing HTML that's in each panel with the new content. (If a change has occurred to content that's not inside an UpdatePanel, it's ignored.)

Note If you access a page that uses the UpdatePanel with a browser that doesn't support Ajax or does JavaScript switched on, the page uses normal postbacks instead of partial updates. However, everything works correctly.

A Simple UpdatePanel Test

To try out the UpdatePanel, it makes sense to build a simple test page. The following example (Figure 25-2) uses a simple page that includes two ingredients: an animated GIF image, and a shaded box that shows the current time and includes a Refresh Time button. When you click the Refresh Time button, the page grabs the current time from the web server and updates the label. However, the refresh process uses a partial update. As a result, the page won't flicker when the update takes place. The animated GIF helps illustrate the point—even as you click the button to refresh the label, the lamp continues bubbling without a pause or hiccup.

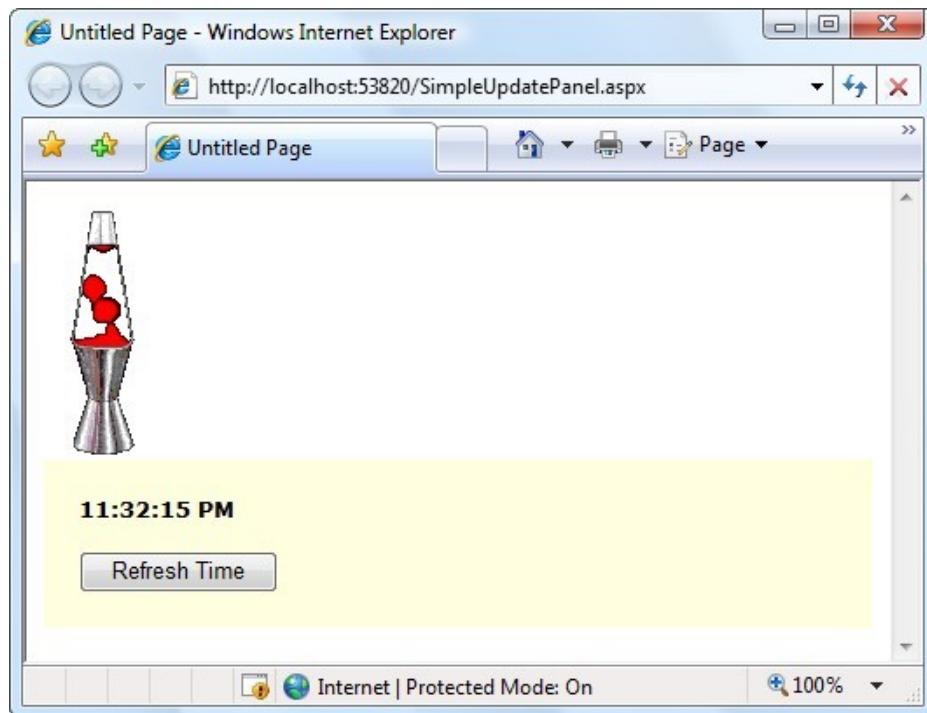


Figure 25-2. Refreshing a label with a partial update

Building this page is easy. First you need to add the ScriptManager control to the page. Like all controls that use ASP.NET AJAX, the UpdatePanel works in conjunction with the ScriptManager. If you don't have it in your page, the UpdatePanel won't work (and you'll receive an exception when you run the page). Furthermore, the ScriptManager needs to appear before the UpdatePanel, because the rendered page must have the JavaScript script block in place before the UpdatePanel can use it. It's a good idea to always place the ScriptManager at the top of the <form> section of the page.

Next you need to add the content to the page. The animated GIF is fairly straightforward—you can use an ordinary `` element to show it. However, the label and button require a bit more effort. To refresh the label by using a partial refresh, you need to wrap it in an `UpdatePanel`. So start by adding an `UpdatePanel` to your page, and then drag and drop the button and label inside.

The `UpdatePanel` has one role in life—to serve as a container for content that you want to refresh asynchronously. Interestingly enough, the `UpdatePanel` doesn't derive from `Panel`. Instead, it derives directly from `Control`. This design causes a few quirks that you should take into account.

First, the `UpdatePanel` is invisible. Unlike the standard ASP.NET Panel, an `UpdatePanel` doesn't support style settings. If you want to display a border around your `UpdatePanel` or change the background color, you'll need to place an ordinary `Panel` (or just a plain `<div>` tag) in your `UpdatePanel`. Here's how it's done in the example shown in Figure 25-2:

```
<asp:ScriptManager ID = "ScriptManager1" runat = "server" ></asp:ScriptManager>
<asp:UpdatePanel ID = "UpdatePanel1" runat = "server" UpdateMode = "Conditional">
<ContentTemplate>
<div style = "background-color:LightYellow;padding: 20px">
<asp:Label ID = "lblTime" runat = "server" Font-Bold = "True" ></asp:Label>
<br />
<br />
<asp:Button ID = "cmdRefreshTime" runat = "server"
OnClick = "cmdRefreshTime_Click"
Text = "Refresh Time" />
</div>
</ContentTemplate>
</asp:UpdatePanel>
```

This markup reveals another difference between the `UpdatePanel` and an ordinary `Panel`—the `UpdatePanel` uses a template. All the controls you add to an `UpdatePanel` are placed in an element named `<ContentTemplate>`. When the `UpdatePanel` renders itself, it copies the content from the `ContentTemplate` into the page. This seems like a fairly unimportant, low-level detail, but it does have one important side effect. If you want to use code to dynamically add controls to an `UpdatePanel`, you can't use the `UpdatePanel.Controls` collection. Instead, you need to add new controls to the `UpdatePanel.ContentTemplateContainer.Controls` collection.

Now that you have the controls you need, you're ready to add the code. This part is easy—when the button is clicked, you simply react to the `Click` event and update the label:

```
protected void cmdRefreshTime_Click(object sender, EventArgs e)
{
    lblTime.Text = DateTime.Now.ToString("T");
}
```

Remarkably, that's all you need to do to complete this example. Now when you click the Refresh Time button, the label will refresh without a full postback and without any flicker.

So how does it all work? Here's a blow-by-blow analysis of what's taking place:

1. When rendering the HTML, the `UpdatePanel` looks at its contents and notices that it contains one control that can trigger a postback—the button. The `UpdatePanel` adds some JavaScript code that will intercept the button's `Click` event on the client and use a JavaScript routine to handle it.
2. When you click the Refresh Time button, you trigger the JavaScript routine.
3. The JavaScript routine doesn't perform a full-page postback. Instead, it sends a background request to the web server. This request is asynchronous, which means your page remains responsive while the request is under way.

Note Because the UpdatePanel uses asynchronous requests, it's possible to click the Refresh Time button several times before the result is returned and the time is updated. In this case, the response from the first request is ignored, and the response from the last request is used. (It's similar to what happens if you refresh a posted-back page several times before it's finished being processed on the server.)

4. The background request is processed in *exactly* the same way as a normal postback. All the data from all the web controls is sent back to the web server, along with the view state information and any cookies. On the web server, the page life cycle is the same—first the Page.Load event fires, followed by the event that triggered the postback (in this case, Button.Click). If you're using data source controls such as SqlDataSource, all the normal querying and data binding takes place. The final page is then rendered to HTML and sent back to the page.
5. When the browser receives the rendered HTML for the page, it updates the current view state and grabs any cookies that were returned.
6. The JavaScript routine then replaces a portion of the HTML on the page—just the portion that you wrapped in the UpdatePanel. The rest of the HTML for the page is simply discarded. In the current example, that means the HTML with the animated GIF is tossed out. (This really has no effect, because this part of the HTML is exactly the same in the new response as it was originally. However, it's important to understand that if you modify this part of your page on the web server, you won't see the results of your change in the web browser, because that area of the page isn't being updated.)

The most impressive aspect of the UpdatePanel control is that it allows your web page to behave in the same way it would if you weren't using any Ajax techniques. There is a bit of a price to pay for this convenience—namely, the request might take a little longer than necessary because of all the extra work that's taking place. In a more streamlined do-it-yourself approach, you'd simply ask the web server for exactly what you need. In this example, that means you'd simply ask for the current time, rather than an entire HTML document.

However, in most scenarios, the UpdatePanel's more long-winded approach doesn't introduce any noticeable delay. Even better, it gives you the ability to deal with much more complex scenarios—for example, when you're modifying a section of a web page much more dramatically. You'll see an example later in this chapter, when you use the UpdatePanel to improve the greeting card maker.

Note When you use the UpdatePanel, you don't reduce the amount of bandwidth being used or the time to receive the response from the server, because the entire page is still sent. The only difference is that the page is updated without a distracting flicker. Small as that advantage seems, it can make a major difference in how the web page "feels" to the person using it.

Error Handling

As you've seen, when the UpdatePanel performs its callback, the web page code runs in exactly the same way as if the page had been posted back. The only difference is the means of communication (the page uses an asynchronous call to get the new data) and the way the received data is dealt with (the UpdatePanel refreshes its inner content, but the remainder of the page is not changed). Therefore, you don't need to make significant changes to your server-side code or deal with new error conditions.

That said, problems can occur when performing an asynchronous postback just as they do when performing a synchronous postback. To find out what happens, you can add code like this to the event handler for the Page_Load event, which causes an unhandled exception to occur when an asynchronous callback takes place:

```
if(this.IsPostBack)
{
    throw new ApplicationException("This operation failed.");
```

When the web page throws an unhandled exception, the error is caught by the ScriptManager and passed back to the client. The client-side JavaScript then throws a JavaScript error. What happens next depends on your browser settings, but usually browsers are configured to quietly suppress JavaScript errors. In Internet Explorer, an Error on Page message appears in the status bar that indicates the problem. If you double-click this notification, a dialog box appears with the full details, as shown in Figure 25-3.

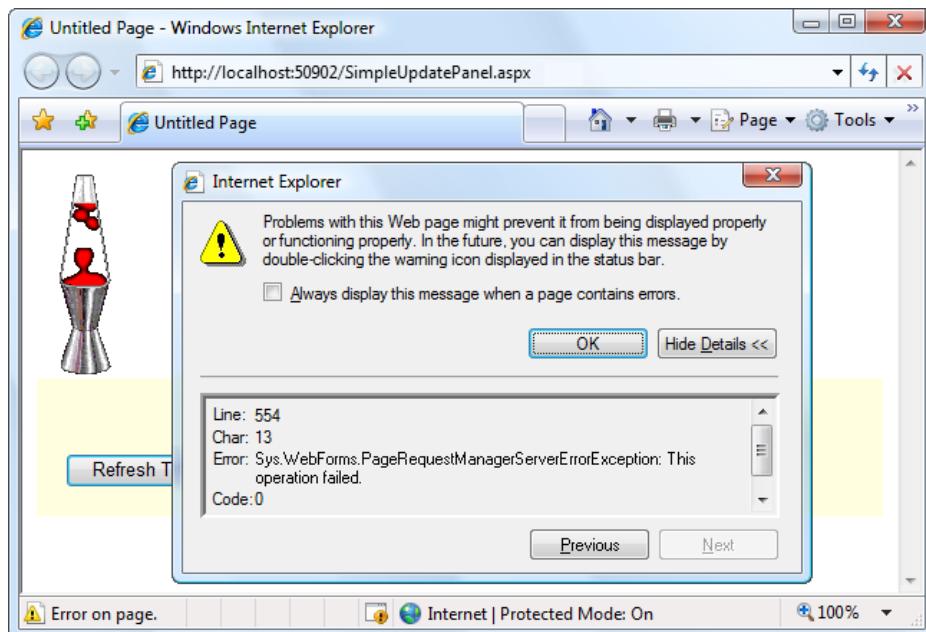


Figure 25-3. Displaying a client-side message about a server-side error

There's another option for dealing with the errors that occur during an asynchronous postback. You can use custom error pages, just as you do with ordinary web pages. All you need to do is add the <customErrors> element to the web.config file.

For example, here's a <customErrors> element that redirects all errors to the page named ErrorPage.aspx:

```
<configuration>
  ...
<system.web>
  ...
  <customErrors defaultRedirect = "ErrorPage.aspx" mode = "On" ></customErrors>
</system.web>
</configuration>
```

Now when the PageRequestManager is informed of an error, it will redirect the browser to `ErrorPage.aspx`. It also adds an `aspxerrorpath` query string argument to the URL that indicates the URL of the page where the problem originated, as shown here:

`http://localhost/Ajax/ErrorPage.aspx?aspxerrorpath=/Ajax/UpdatePanels.aspx`

You can write code in `ErrorPage.aspx` that reads the `aspxerrorpath` information. For example, you might include a Retry button that redirects the user to the original requested page, like this:

```
string url = Request.QueryString["aspxerrorpath"];
if (url != null) Response.Redirect(url);
```

If your website uses custom error pages but you don't want them to apply to asynchronous postbacks, you must set the `ScriptManager.AllowCustomErrorsRedirect` property to false.

Note Several controls don't work correctly in an `UpdatePanel` or don't work in specific scenarios. Most the `FileUpload` and `HtmlInputFile` controls don't work at all. The `Login`, `PasswordRecovery`, `ChangePassword`, `CreateUserWizard` controls work only if you've converted their content to templates. The `GridView` and `DetailView` controls fail if you've set `EnableSortingAndPagingCallbacks` to true. And the `TreeView` and `Menu` controls fail if you've set their style properties (instead, you can format their items by using CSS styles). However, all the other controls will work on pages that contain `UpdatePanel` controls, so long as they aren't actually in an `UpdatePanel`.

Conditional Updates

In complex pages, you might have more than one `UpdatePanel`. In this case, when one `UpdatePanel` triggers an update, all the `UpdatePanel` regions will be refreshed.

However, if you have more than one `UpdatePanel` and each one is completely self-contained, this isn't necessary. In this situation, you can configure the panels to update themselves independently. Simply change the `UpdatePanel.UpdateMode` property from `Always` to `Conditional`. Now the `UpdatePanel` will refresh itself only if an event occurs in one of the controls in that `UpdatePanel`.

To try this out, create a page that has several `UpdatePanel` controls, each with its own time display and button. Then add code that places the current time in the label of all three controls:

```
protected void Page_Load(object sender, EventArgs e)
{
    lblTime1.Text = DateTime.Now.ToString("T");
    lblTime2.Text = DateTime.Now.ToString("T");
    lblTime3.Text = DateTime.Now.ToString("T");
}
```

When you click one of the Refresh Time buttons, only the label in that panel will be updated. The other panels will remain untouched.

Note There's an interesting quirk here. Technically, when you click the button, all the labels are updated, but only part of the page is refreshed to show that fact. The next time the page is posted back, the most recently updated controls are pulled out of view state and applied to all the labels, including the ones that weren't refreshed on the previous postback.

Most of the time, this distinction isn't important. But if this isn't the behavior you want in this example, you can add a separate event handler for each button. Each event handler would update just one label—the label that's contained within the appropriate UpdatePanel. That way, when the page is posted back, just one label is changed, and you don't have to worry about updating time changing parts of the page that won't be updated in the browser.

There's one caveat with this approach. If you perform an update that takes a long time, it could be interrupted by another update. As you know, ASP.NET AJAX posts the page back asynchronously, so the user is free to click other buttons while the postback is under way. ASP.NET AJAX doesn't allow concurrent updates, because it needs to ensure that other information—such as the page view state information, the session cookie, and so on—remains consistent. Instead, when a new asynchronous postback is started, the previous asynchronous postback is abandoned. For the most part, this is the behavior you want. If you want to prevent the user from interrupting an asynchronous postback, you can add JavaScript code that disables controls while the asynchronous postback is under way, but this takes significantly more work. Another option is to use the UpdateProgress control discussed later in this chapter.

Tip There's one other way to update a conditional UpdatePanel—you can call the `UpdatePanel.Update` method from your server-side code. This allows you to decide on the fly whether a certain panel should be refreshed. However, you must be careful not to call `Update()` on a panel that uses an `UpdateMode` of `Always`, and you mustn't call `Update()` after the page has been rendered. If you make either of these mistakes, you'll cause an exception.

Triggers

So far, the examples you've seen have used the built-in behavior of the `UpdatePanel`. In the next example, you'll use the `UpdatePanel` in a slightly more sophisticated page, and you'll learn to control its refreshes with triggers.

Earlier in this book, you learned how to build a web page that allows the user to dynamically build a greeting card. This page lets the user specify several details—such as text, font, colors, border options, and so on—and then updates a portion of the page to show the greeting card.

Chapter 6 demonstrated two versions of the greeting card maker. The first version allowed the user to specify a number of options at once, and then click a button to update the greeting card. The second version used automatic postback events, so that the greeting card was updated after every change. This second approach gave a more immediate result, but the cost was a less responsive user interface with distracting flicker. If this version of the greeting card maker was running on a slow web server (or over a slow network), the problems get even worse. That's because after every postback, the user is forced to wait for the page update before making another change.

The `UpdatePanel` gives you the ability to get the best of both versions. You can create a greeting card page that updates its display automatically but feels more responsive and doesn't lock the user out.

The simplest approach is to add a `ScriptManager` and wrap the entire web page—including the controls and the greeting card—in one giant `UpdatePanel`. Here's the markup you'd use:

```
<asp:ScriptManager ID = "ScriptManager1" runat = "server" > </asp:ScriptManager>
<asp:UpdatePanel ID = "UpdatePanel1" runat = "server">
<ContentTemplate>
    <!-- These are the controls for creating the greeting card. -->
    <div style = "...>
        Choose a background color:<br />
        <asp:DropDownList ID = "lstBackColor" runat = "server" >
```

```

Width = "194px" AutoPostBack = "True">
</asp:DropDownList>
<br /> <br />
Choose a foreground (text) color:<br />
<asp:DropDownList ID = "lstForeColor" runat = "server"
Height = "22px" Width = "194px" AutoPostBack = "True" >
</asp:DropDownList>
<br /> <br />
Choose a font name:<br />
<asp:DropDownList ID = "lstFontName" runat = "server"
Height = "22px" Width = "194px" AutoPostBack = "True">
</asp:DropDownList>
<br /> <br />
Specify a font size:<br />
<asp:TextBox ID = "txtFontSize" runat = "server"
AutoPostBack = "True">
</asp:TextBox>
<br /> <br />
Choose a border style:<br />
<asp:RadioButtonList ID = "lstBorder" runat = "server"
Height = "59px" Width = "177px" Font-Size = "X-Small"
AutoPostBack = "True" RepeatColumns = "2">
</asp:RadioButtonList>
<br /> <br />
<asp:CheckBox ID = "chkPicture" runat = "server"
Text = "Add the Default Picture" AutoPostBack = "True">
</asp:CheckBox>
<br /> <br />
Enter the greeting text below:<br />
<asp:TextBox ID = "txtGreeting" runat = "server"
Height = "85px" Width = "240px" TextMode = "MultiLine"
AutoPostBack = "True">
</asp:TextBox>
</div>

<!-- This is the panel that shows the greeting card. -->
<asp:Panel ID = "pnlCard" runat = "server" ... >
<asp:Label ID = "lblGreeting" runat = "server" Width = "272px"
Height = "150px" > </asp:Label>
<br />
<asp:Image ID = "imgDefault" runat = "server" Width = "212px" Height = "160px"
Visible = "False" > </asp:Image>
</asp:Panel>

</ContentTemplate>
</asp:UpdatePanel>

```

The greeting card is then generated when the Page.Load event fires:

```

protected void Page_Load(object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {

```

```
// (Initialize all the controls here.)
}
else
{
    // Refresh the greeting card.
    UpdateCard();
}
```

The UpdatePanel watches its child controls and intercepts any events that could trigger a postback. The Button.Click event is an obvious example, but in this instance the TextBox.TextChanged and ListBox.SelectedIndexChanged events also trigger a postback, because these controls set the AutoPostBack property to true. Thus, these events are also intercepted by the UpdatePanel. If these controls didn't use the AutoPostBack property, they wouldn't trigger a postback, the UpdatePanel wouldn't get involved, and the greeting card wouldn't be updated until another control caused a postback.

This solution achieves the desired result. Although the greeting card page looks essentially the same (see Figure 25-4), when you interact with the controls on the left, the card on the right is updated without a postback.

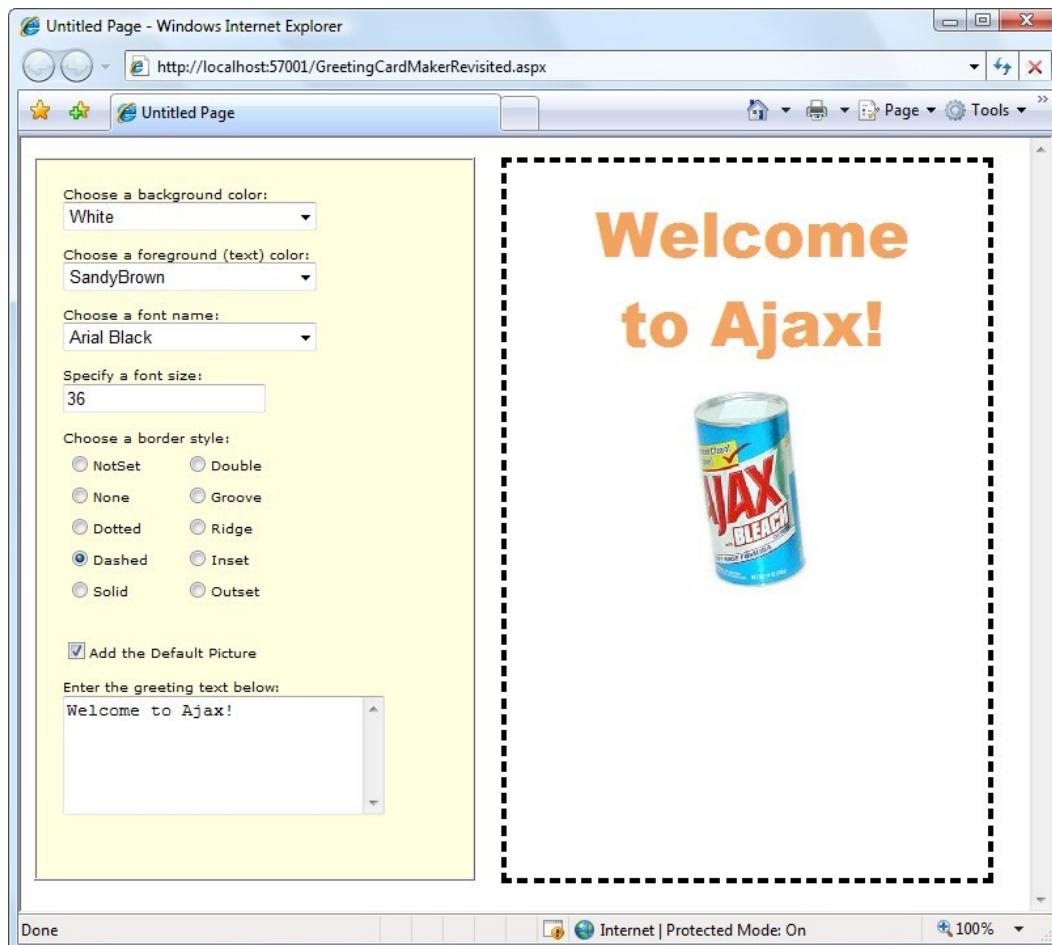


Figure 25-4. The greeting card maker, Ajax style

If you make several changes in quick succession, you'll trigger several postbacks, and the result from the last postback (with the completely updated card) will be used.

Although this example works perfectly well, it's doing more work than necessary. Because the entire page is wrapped in the UpdatePanel, the HTML for the entire page is refreshed. A better option is to wrap just the label and image that represents the greeting card in the UpdatePanel. Unfortunately, this won't work. After you move the other controls out of the UpdatePanel, their events won't be intercepted any longer, and they'll trigger full-page postbacks with the familiar flicker.

The solution is to explicitly tell the UpdatePanel to monitor those controls, even though they aren't inside the UpdatePanel. You can do this by adding triggers to the UpdatePanel. You add one trigger for each button.

Here's the markup you need:

```
<asp:ScriptManager ID = "ScriptManager1" runat = "server" ></asp:ScriptManager>
<!-- The controls for creating the greeting card go here. -->
<asp:UpdatePanel ID = "UpdatePanel1" runat = "server">
<ContentTemplate>
<!-- This is the panel that shows the greeting card. -->
<asp:Panel ID = "pnlCard" runat = "server" ... >
<asp:Label ID = "lblGreeting" runat = "server" Width = "272px"
Height = "150px" ></asp:Label>
<asp:Image ID = "imgDefault" runat = "server" Width = "212px" Height = "160px"
Visible = "False" ></asp:Image>
</asp:Panel>
</ContentTemplate>
<Triggers>
<asp:AsyncPostBackTrigger ControlID = "lstBackColor" />
<asp:AsyncPostBackTrigger ControlID = "lstForeColor" />
<asp:AsyncPostBackTrigger ControlID = "lstFontName" />
<asp:AsyncPostBackTrigger ControlID = "lstFontSize" />
<asp:AsyncPostBackTrigger ControlID = "lstBorder" />
<asp:AsyncPostBackTrigger ControlID = "chkPicture" />
<asp:AsyncPostBackTrigger ControlID = "txtGreeting" />
</Triggers>
</asp:UpdatePanel>
```

Tip You don't need to type your triggers in by hand. Instead, you can use the Visual Studio Properties window. Just select the UpdatePanel, click the Triggers property in the Properties window, and click the ellipsis (...) button that appears in the Triggers box. Visual Studio will open a dialog box where you can add as many triggers as you want and pick the control for each trigger from a drop-down list.

These triggers tell the UpdatePanel to intercept the default event in these seven controls. (You could specify the event you want to monitor by setting the EventName property of the trigger, but you don't need to, because you're using the most commonly used default event for each one.) As a result, the page works the same as it did before—it just refreshes a smaller portion of the page after each asynchronous request.

Technically, the UpdatePanel always uses triggers. All the controls inside an UpdatePanel automatically become the triggers for the UpdatePanel. However, you need to add the triggers if the controls are placed elsewhere in the page.

Note You can add the same trigger to several different conditional UpdatePanel controls, in which case the event will update them all.

You can use triggers in one other way. Instead of using them to monitor more controls, you can use them to tell the UpdatePanel to ignore certain controls. For example, imagine you have a button in your UpdatePanel. Ordinarily, clicking that button will trigger an asynchronous request and partial update. If you want it to trigger a full-page postback instead, you simply need to add a PostBackTrigger (instead of an AsynchronousPostBackTrigger).

For example, here's an UpdatePanel that contains a nested button that triggers a full postback rather than an asynchronous postback:

```
<asp:UpdatePanel ID = "UpdatePanel1" runat = "server" UpdateMode = "Conditional">
<ContentTemplate>
<asp:Label ID = "Label1" runat = "server" Font-Bold = "True" > </asp:Label>
<br />
<br />
<asp:Button ID = "cmdPostback" runat = "server" Text = "Refresh Full Page" />
</ContentTemplate>
<Triggers>
<asp:PostBackTrigger ControlID = "cmdPostback" />
</Triggers>
</asp:UpdatePanel>
```

This technique isn't as common, but it can be useful if you have several controls in an UpdatePanel that perform limited updates (and so use asynchronous request) and one that performs more-significant changes to the whole page (and so uses a full-page postback).

Using Progress Notification

As you've learned, the UpdatePanel performs its work asynchronously in the background. As a result, the user can keep working with the page. This is generally the behavior you want, but there's one catch. While the asynchronous request is under way, the user won't necessarily realize that anything's happening. If the asynchronous request takes some time, this could be a bit of a problem. At worst, the user will assume the page is broken or click the same button multiple times, creating needless extra work for your web application and slowing down the process further.

ASP.NET includes another control that can help—the UpdateProgress control. The UpdateProgress control works in conjunction with the UpdatePanel. Essentially, the UpdateProgress control allows you to show a message while a time-consuming update is under way.

Note The UpdateProgress control is slightly misnamed. It doesn't actually indicate progress; instead, it displays a wait message that reassures the user that the page is still working and the last request is still being processed.

Showing a Simulated Progress Bar

When you add the UpdateProgress control to a page, you get the ability to specify some content that will appear as soon as an asynchronous request is started and will disappear as soon as the request is finished. This content can include a fixed message, but many people prefer to use an animated GIF, because it more clearly suggests that the page is still at work. Often this animated GIF simulates a progress bar.

Figure 25-5 shows a page that uses the UpdateProgress control at three points in its life cycle. The top image shows the page as it first appears, with a straightforward UpdatePanel control containing a button. When the button is clicked, the asynchronous callback process begins. At this point, the contents of the UpdateProgress control appear underneath (as shown in the middle image). In this example, the UpdateProgress includes a text

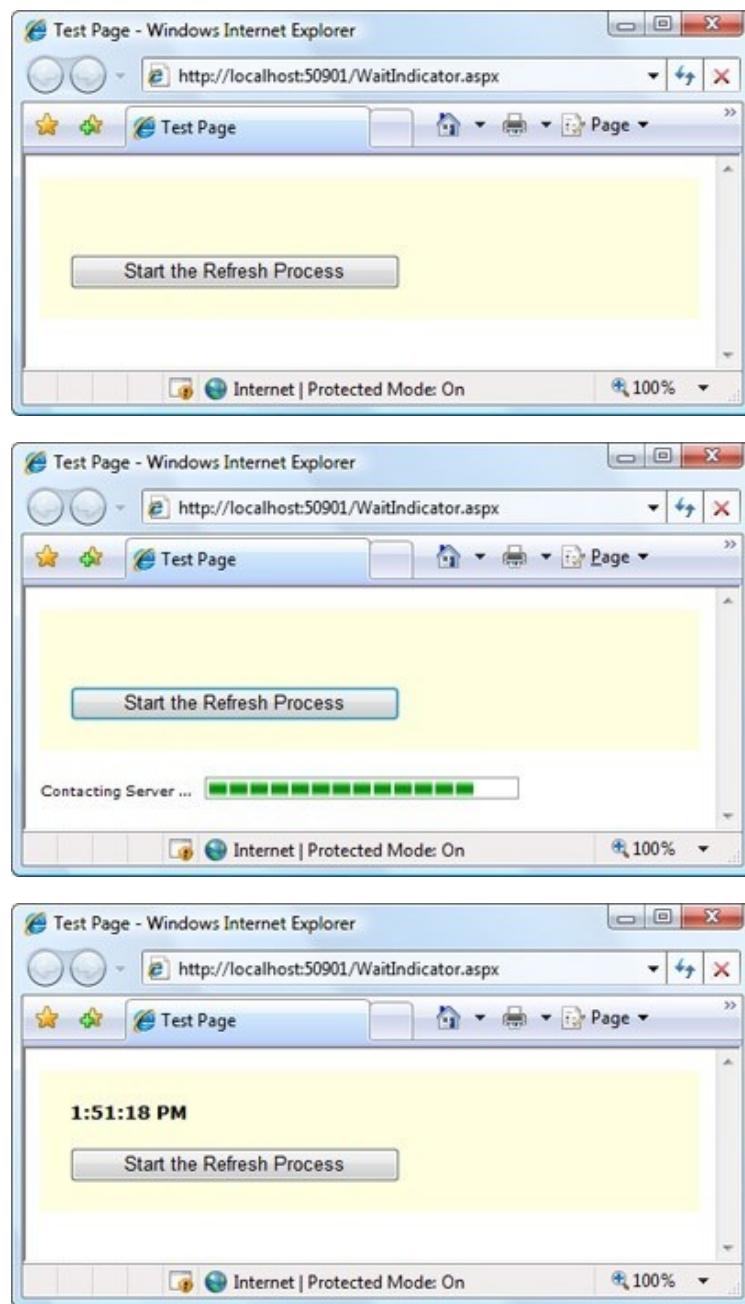


Figure 25-5. A wait indicator

message and an animated GIF that appears as a progress bar, with green blocks that perpetually fill it from left to right, and then start over. When the callback is complete, the UpdateProgress disappears and the UpdatePanel is updated, as shown in the bottom image.

The markup for this page defines an UpdatePanel followed by an UpdateProgress:

```
<asp:UpdatePanel ID = "UpdatePanel1" runat = "server">
<ContentTemplate>
<div style = "background-color:#FFFFE0;padding: 20px">
<asp:Label ID = "lblTime" runat = "server" Font-Bold = "True" ></asp:Label>
<br /> <br />
<asp:Button ID = "cmdRefreshTime" runat = "server"
    OnClick = "cmdRefreshTime_Click"
    Text = "Start the Refresh Process" />
</div>
</ContentTemplate>
</asp:UpdatePanel>
<br />

<asp:UpdateProgress ID = "updateProgress1" runat = "server">
<ProgressTemplate>
<div style = "font-size: xx-small">
    Contacting Server ... <img src = "wait.gif" alt = "Waiting..." />
</div>
</ProgressTemplate>
</asp:UpdateProgress>
```

This isn't the only possible arrangement. Depending on the layout you want, you can place your UpdateProgress control somewhere inside your UpdatePanel control.

The code for this page has a slight modification from the earlier examples. Because the UpdateProgress control shows its content only while the asynchronous callback is under way, it makes sense to use the control only with an operation that takes time. Otherwise, the UpdateProgress will show its ProgressTemplate for only a few fractions of a second. To simulate a slow process, you can add a line to delay your code 10 seconds, as shown here:

```
protected void cmdRefreshTime_Click(object sender, EventArgs e)
{
    System.Threading.Thread.Sleep(TimeSpan.FromSeconds(10));
    lblTime.Text = DateTime.Now.ToString();
}
```

There's no need to explicitly link the UpdateProgress control to your UpdatePanel control. The UpdateProgress automatically shows its ProgressTemplate whenever *any* UpdatePanel begins a callback. However, if you have a complex page with more than one UpdatePanel, you can choose to limit your UpdateProgress to pay attention to just one of them. To do so, simply set the UpdateProgress.AssociatedUpdatePanelID property with the ID of the appropriate UpdatePanel. You can even add multiple UpdateProgress controls to the same page, and link each one to a different UpdatePanel.

Enabling Cancellation

The UpdateProgress control supports one other detail: a cancel button. When the user clicks a cancel button, the asynchronous callback will be cancelled immediately, the UpdateProgress content will disappear, and the page will revert to its original state.

Adding a cancel button is a two-step process. First you need to add a fairly intimidating block of JavaScript code, which you can copy verbatim. You should place this code at the end of your page, after all your content but just before the </body> end tag. Here's the code you need, in its rightful place:

```
<%@ Page Language = "C#" AutoEventWireup = "true" CodeFile = "WaitIndicator.aspx.cs"
Inherits = "WaitIndicator" %>
<!DOCTYPE html>
<html>
<head runat = "server">
...
</head>
<body>
<form ID = "form1" runat = "server">
...
</form>

<script type = "text/javascript">
var prm = Sys.WebForms.PageRequestManager.getInstance();
prm.add_initializeRequest(InitializeRequest);

function InitializeRequest(sender, args)
{
  if(prm.get_isInAsyncPostBack())
  {
    args.set_cancel(true);
  }
}

function AbortPostBack()
{
  if(prm.get_isInAsyncPostBack())
  {
    prm.abortPostBack();
  }
}
</script>
</body>
</html>
```

Tip You can cut and paste this code from the WaitIndicator.aspx page (which is included with the sample code for this chapter) into your own pages.

Once you've added this code, you can use JavaScript code to call its AbortPostBack() function at any time and cancel the callback. The easiest way to do this is to connect a JavaScript event to the AbortPostBack() function using a JavaScript event attribute. You can add a JavaScript event attribute to virtually any HTML element. For example, you can deal with client-side clicks by using the onclick attribute. Here's a basic HTML button (not a server control) that uses this technique to connect itself to the AbortPostBack() function:

```
<input ID = "cmdCancel" onclick = "AbortPostBack()" type = "button" value = "Cancel" />
```

If you click this Cancel button, the client-side AbortPostBack() function will be triggered and the callback will be cancelled immediately. Typically, you'll place this button (or an element like this) in the ProgressTemplate of the UpdateProgress control, as shown in Figure 25-6.

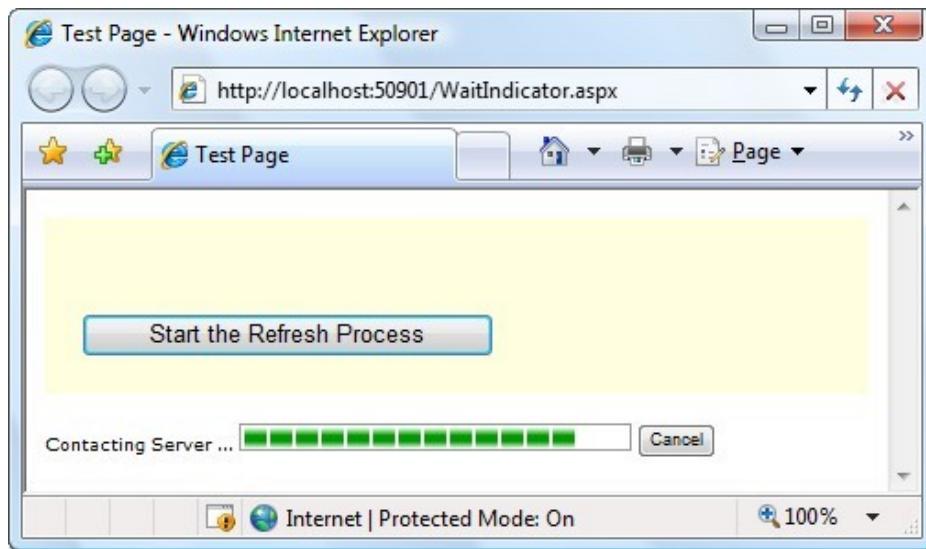


Figure 25-6. An UpdateProgress control with a cancel button

Don't confuse this approach with server-side event handling—the client-side onclick attribute allows you to intercept an event in the browser and process it by using JavaScript code. The server doesn't get involved at all. In fact, when you cancel an operation, the server continues to process the request, but the browser simply closes the connection and stops listening for the response.

Tip It makes sense to use an abort button for tasks that can be safely canceled because they don't change external state. For example, users should be able to cancel an operation that retrieves information from a database. However, it's not a good idea to add cancellation to an operation that updates a database, because the server will continue until it finishes the update, even if the client has stopped listening for the response.

Implementing Timed Refreshes

Using the two controls you've seen so far—the UpdatePanel and UpdateProgress controls—you can create self-contained regions on your page that refresh themselves when certain actions take place. Of course, in order for this technique to work, the user needs to initiate an action that would ordinarily cause a postback, such as clicking a button, selecting an item in an AutoPostBack list, selecting an AutoPostBack check box, and so on.

In some situations, you might want to force a full- or partial-page refresh without waiting for a user action. For example, you might create a page that includes a stock ticker, and you might want to refresh this ticker periodically (say, every five minutes) to ensure that it doesn't become drastically outdated. ASP.NET includes a Timer control that allows you to implement this design easily.

The Timer control is refreshingly straightforward. You simply add it to a page and set its Interval property to the maximum number of milliseconds that should elapse before an update. For example, if you set Interval to 60000, the timer will force a postback after one minute elapses.

```
<asp:Timer ID = "Timer1" runat = "server" Interval = "60000" />
```

Note Obviously, the timer has the potential to greatly increase the overhead of your web application and its scalability. Think carefully before introducing timed refreshes, and make the intervals long rather than short.

The Timer control works by rendering a bit of client-side script that starts a JavaScript timer. When the JavaScript timer fires (at the interval you specify), the client-side script triggers a postback and raises a server-side Tick event.

On its own, this behavior is fairly inefficient (because it causes the page to completely refresh itself, possibly while the user is in the midst of another task). But if you place the Timer in an UpdatePanel, the result is much better. The UpdatePanel converts the postback to a seamless callback and uses partial rendering to update just part of the page. Unlike a full postback, a callback with partial rendering won't cause flicker and won't interrupt the user in the middle of a task.

To use the timer with partial rendering, wrap the updateable portions of the page in UpdatePanel controls with the UpdateMode property set to Conditional. Then add a trigger that forces the UpdatePanel to update itself whenever the Timer.Tick event occurs. Here's the markup you need:

```
<asp:UpdatePanel ID = "UpdatePanel1" runat = "server" UpdateMode = "Conditional">
<ContentTemplate>
...
</ContentTemplate>
<Triggers>
<asp:AsyncPostBackTrigger ControlID = "Timer1" EventName = "Tick" />
</Triggers>
</asp:UpdatePanel>
```

All the other portions of the page can be left as is, or you can wrap them in conditional UpdatePanel controls with different triggers if you need to update them in response to other actions.

Note You must use triggers with the Timer control. You can't simply place the timer inside an UpdatePanel and expect it to work without a trigger (unlike other controls). If you don't use a trigger, the timer will force a full postback, with flicker.

To stop the timer, you simply need to set the Enabled property to false in server-side code. For example, here's how you could disable the timer after ten updates:

```
protected void Timer1_Tick(object sender, EventArgs e)
{
    // Update the tick count and store it in view state.
    int tickCount = 0;
    if (ViewState["TickCount"] != null)
    {
        tickCount = (int)ViewState["TickCount"];
    }
    tickCount++;
    ViewState["TickCount"] = tickCount;

    // Decide whether to disable the timer.
    if (tickCount > 10)
    {
        Timer1.Enabled = false;
    }
}
```

Working with the ASP.NET AJAX Control Toolkit

The UpdatePanel, UpdateProgress, and Timer controls are fairly useful. However, they're the only ASP.NET AJAX-enabled controls you'll find in ASP.NET. Despite their value, developers who have heard the Ajax hype and used advanced Ajax websites such as Gmail might expect a bit more.

In fact, ASP.NET's support for Ajax is a bit deceptive. Although there are only three controls that use ASP.NET AJAX features, ASP.NET includes a sophisticated library of JavaScript functions that can be used to create all sorts of advanced effects. Business application developers aren't likely to use these libraries (because they're quite complex and require a significant time investment), but third-party component developers will use them enthusiastically.

The first and best example of what ASP.NET AJAX can really do is the ASP.NET AJAX Control Toolkit. The ASP.NET AJAX Control Toolkit is a joint project between Microsoft and the ASP.NET community. It consists of dozens of controls that use the ASP.NET AJAX libraries to create sophisticated effects.

The ASP.NET AJAX Control Toolkit has a lot going for it:

- It's completely free.
- It includes full source code, which is helpful if you're ambitious enough to want to create your own custom controls that use ASP.NET AJAX features.
- It uses extenders that enhance the standard ASP.NET web controls. That way, you don't have to replace all the controls on your web pages—instead, you simply plug in the new bits of functionality that you need.

Installing the ASP.NET AJAX Control Toolkit

You can download the ASP.NET AJAX Control Toolkit from <http://ajaxcontroltoolkit.codeplex.com>. Click the Downloads tab, and find the toolkit for the latest version of .NET. After you've downloaded this ZIP file, you can extract the files it contains to a more permanent location on your hard drive.

Note At the time of this writing, the latest build of the ASP.NET AJAX Control Toolkit is named AjaxControlToolkitBinary.NET4.zip and targets .NET 4, but it still works equally well with .NET 4.5. If you need support for AS 4.5, you'll find a similar download that targets that version of .NET.

Inside the ZIP file, you'll find a central assembly named AjaxControlToolkit.dll and a host of smaller satellite assemblies that support localization for different cultures. You'll also see a zipped folder named AjaxControlToolkitSampleSite, which contains a huge sample website that demonstrates all the ASP.NET AJAX Control Toolkit ingredients, and a ReadMe.html file with installation instructions.

To get started developing with the ASP.NET AJAX Control Toolkit, you could simply open the AjaxControlToolkitSampleSite folder (or copy the contents of its Bin subfolder to the Bin subfolder in one of your web applications). However, life is much easier if you get Visual Studio to help you out by adding the new components to the Toolbox. Here's how:

1. Make sure you've put the unzipped contents of the ASP.NET AJAX Control Toolkit in a reasonably permanent location on your hard drive. (For example, you could copy them to a folder such as c:\ASP.NETAjaxControlToolkit.) If you move the folder after you complete this process, Visual Studio won't be able to find the AjaxControlToolkit.dll assembly. As a result, it won't be able to add the necessary assembly reference when you drag the controls onto a web page. (The only way to fix this problem is to remove the controls from the Toolbox and then repeat the process to add them from their new location.)
2. First, you need to create a new Toolbox tab for the controls. Right-click the Toolbox and choose Add Tab. Then enter a name (for example, AJAX Toolkit) and press Enter.
3. Now, you need to add the controls to the new tab. Right-click the blank tab you've created and select Choose Items.
4. In the Choose Toolbox Items dialog box, click Browse. Find the AjaxControlToolkit.dll (which is in the folder where you unzipped the ASP.NET AJAX Control Toolkit) and click OK.
5. Visual Studio may warn you that the DLL was downloaded from the Internet. Click Yes to accept this warning and continue.
6. Now, all the components from AjaxControlToolkit.dll will appear in the list in the Choose Toolbox Items dialog box. Each one is selected, as indicated by a check mark. To add all the controls to the Toolbox in one step, just click OK.

Figure 25-7 shows some of the controls that will appear in the new Toolbox tab.

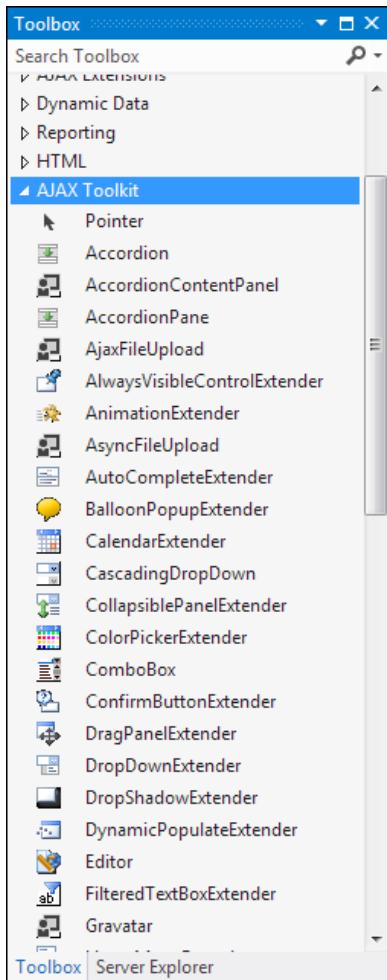


Figure 25-7. Adding the ASP.NET AJAX Control Toolkit to the Toolbox

Now you can use the components from the ASP.NET AJAX Control Toolkit in any web page in any website. Begin by adding the ScriptManager control to the web page. Then head to the new Toolbox tab you created and drag the ASP.NET AJAX control you want onto your page. The first time you add a component from the ASP.NET AJAX Control Toolkit, Visual Studio will copy the AjaxControlToolkit.dll assembly to the Bin folder of your web application, along with the localization assemblies. It will also add a Register directive that maps the controls in AjaxControlToolkit.dll to the namespace prefix *asp* so they're available in your code:

```
<%@ Register Assembly = "AjaxControlToolkit" Namespace = "AjaxControlToolkit"
   TagPrefix = "asp" %>
```

The ASP.NET AJAX Control Toolkit is stuffed full of useful components. In the following sections, you'll get your feet wet by considering just two of the controls you'll find—the Accordion and the AutoCompleteExtender.

Using the Accordion

The Accordion is a container that stacks several panels on top of one another, and allows you to view one at a time. Each panel has a header (which usually displays a section title) and some content. When you click the header of one of the panels, that panel is expanded and the other panels are collapsed, leaving just their headers visible. Figure 25-8 demonstrates the effect you'll see as you click different headers.

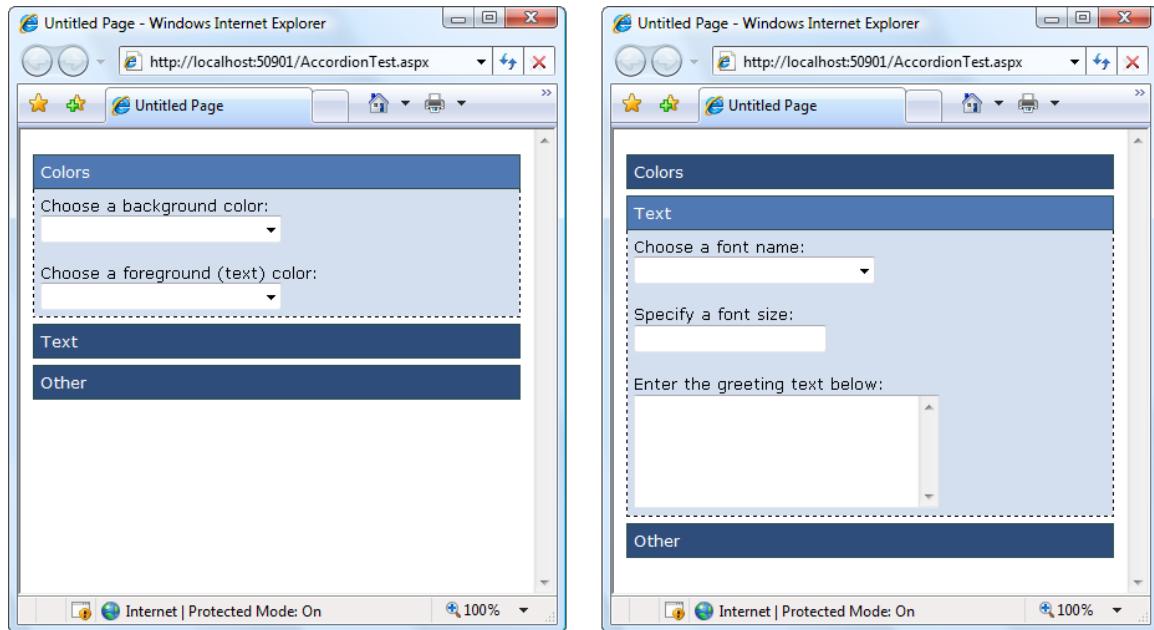


Figure 25-8. Choosing a panel in the Accordion

It goes without saying that the collapsing behavior happens without a full-page postback. In fact, there's no need to contact the web server at all. The first time the page is generated, all the panels are rendered to HTML, but they're hidden by using CSS style attributes. When you click a header, a JavaScript routine runs and changes these style attributes. (In truth, the Accordion control is a bit more sophisticated than that. When you choose a new panel, it gradually *expands* into view, which is much more impressive than simply popping into existence in one step. Furthermore, you can set the *FadeTransitions* property to true if you want panels to fade into and out of sight when you change from one panel to another.)

Using the Accordion control is a bit like using the MultiView control you learned about in Chapter 10. The Accordion control contains a collection of AccordionPane controls. Each AccordionPane represents a separate panel in the Accordion.

Here's an example that illustrates this structure by putting two AccordionPane objects inside the Accordion:

```
<asp:Accordion ID = "Accordion1" runat = "server">
<Panes>
<asp:AccordionPane runat = "server">
...
</asp:AccordionPane>
```

```
<asp:Accordion runat = "server">
...
</asp:Accordion>
</Panes>
</asp:Accordion>
```

Tip To determine what AccordionPane is currently visible (or to set it), you use the Accordion.Selected property. If RequiredOpenedPane is true, there will always be at least one expanded panel. If it's false, you can collapse all the panels—just click the header of the currently expanded section (or set the SelectedIndex to -1 in your code).

Each AccordionPane consists of two sections. The Header section is used for the clickable header of the panel, while the Content holds the details inside. Here's the markup you need to create the example shown in Figure 25-8:

```
<asp:Accordion ID = "Accordion1" runat = "server"
HeaderCssClass = "accordionHeader"
HeaderSelectedCssClass = "accordionHeaderSelected"
ContentCssClass = "accordionContent">

<Panes>
<asp:AccordionPane ID = "paneColors" runat = "server">
<Header > Colors</Header>
<Content>
Choose a background color:<br />
<asp:DropDownList ID = "lstBackColor" runat = "server"
Width = "194px" AutoPostBack = "True">
</asp:DropDownList>
<br /> <br />
Choose a foreground (text) color:<br />
<asp:DropDownList ID = "lstForeColor" runat = "server"
Height = "22px" Width = "194px" AutoPostBack = "True" >
</asp:DropDownList>
</Content>
</asp:AccordionPane>

<asp:AccordionPane ID = "paneText" runat = "server">
<Header > Text</Header>
<Content>
Choose a font name:<br />
<asp:DropDownList ID = "lstFontName" runat = "server"
Height = "22px" Width = "194px" AutoPostBack = "True">
</asp:DropDownList>
<br /> <br />
Specify a font size:<br />
<asp:TextBox ID = "txtFontSize" runat = "server"
AutoPostBack = "True">
</asp:TextBox>
<br /> <br />
Enter the greeting text below:<br />
```

```

<asp:TextBox ID = "txtGreeting" runat = "server"
Height = "85px" Width = "240px" TextMode = "MultiLine"
AutoPostBack = "True">
</asp:TextBox>
</Content>
</asp:AccordionPane>

<asp:AccordionPane ID = "paneExtras" runat = "server">
<Header> Other</Header>
<Content>
Choose a border style:<br />
<asp:RadioButtonList ID = "lstBorder" runat = "server"
Height = "59px" Width = "177px" Font-Size = "X-Small"
AutoPostBack = "True" RepeatColumns = "2">
</asp:RadioButtonList>
<br /> <br />
<asp:CheckBox ID = "chkPicture" runat = "server"
Text = "Add the Default Picture" AutoPostBack = "True">
</asp:CheckBox>
</Content>
</asp:AccordionPane>
</Panes>
</asp:Accordion>

```

Along with the content, this example adds three properties: HeaderCssClass, HeaderSelectedCssClass, and ContentCssClass. These properties take the names of CSS styles that the Accordion uses to format the appropriate region. The styles are defined in a separate style sheet, and look like this:

```

.accordionHeader
{
    border: 1px solid #2F4F4F;
    color: white;
    background-color: #2E4d7B;
    padding: 5px;
    margin-top: 5px;
    cursor: pointer;
}

.accordionHeaderSelected
{
    border: 1px solid #2F4F4F;
    color: white;
    background-color: #5078B3;
    padding: 5px;
    margin-top: 5px;
    cursor: pointer;
}

.accordionContent
{
    background-color: #D3DEEF;
    border: 1px dashed;
    border-top: none;
    padding: 5px;
}

```

Chapter 12 has more about styles. You don't need to use them with the Accordion—after all, you could just set the formatting properties of your controls, or wrap each separate section in a formatted Panel or <div> element—but the style approach is pretty convenient once you get used to it.

You can really fine-tune the display that the Accordion uses by setting a few more properties. TransitionDuration sets the number of milliseconds that the collapsing and expanding animation lasts. FramesPerSecond controls how smooth the transition animation is—a higher value produces better quality, but requires more work from the browser. Finally, AutoSize lets you control how the Accordion expands when you show a panel with a large amount of content. Use a value of None to let the Accordion grow as large as it needs (in which case other content underneath the Accordion is simply bumped out of the way). Use Limit or Fill to restrict the Accordion to whatever you've set in the Height property (the difference is that the Limit allows the Accordion to shrink smaller, while Fill keeps it at the maximum height by leaving any unused space blank). With Limit or Fill, the Accordion panels will use scrolling if they can't fit all their content into the available space.

Clearly, the Accordion is a simple-to-use yet impressive way to deal with dense displays of data and groups of information. If you want to have a similar collapsing and expanding effect with a single panel, you might want to try another one of the components in the ASP.NET AJAX Control Toolkit—the CollapsiblePanelExtender.

Using the AutoCompleteExtender

The Accordion is an example of a completely new control that has ASP.NET AJAX features baked in. Although this is a perfectly reasonable approach to integrating Ajax techniques with the web control model, it's not the most common solution that you'll see used in the ASP.NET AJAX Control Toolkit. In fact, the toolkit includes just a few new controls, and a much larger set of control extenders.

A *control extender* is a .NET component that adds features to an existing ASP.NET control. Control extenders allow you to use the same Ajax effects with countless different controls. This is useful when you're dealing with multipurpose features such as automatic completion, drag-and-drop, animation, resizing, collapsing, masked editing, and so on.

One of the many control extenders in the ASP.NET AJAX Control Toolkit is the AutoCompleteExtender, which allows you to show a list of suggestions while a user types in another control (such as a text box). Figure 25-9 shows the AutoCompleteExtender at work on an ordinary TextBox control. As the user types, the drop-down list offers suggestions. If the user clicks one of these items in the list, the corresponding text is copied to the text box.

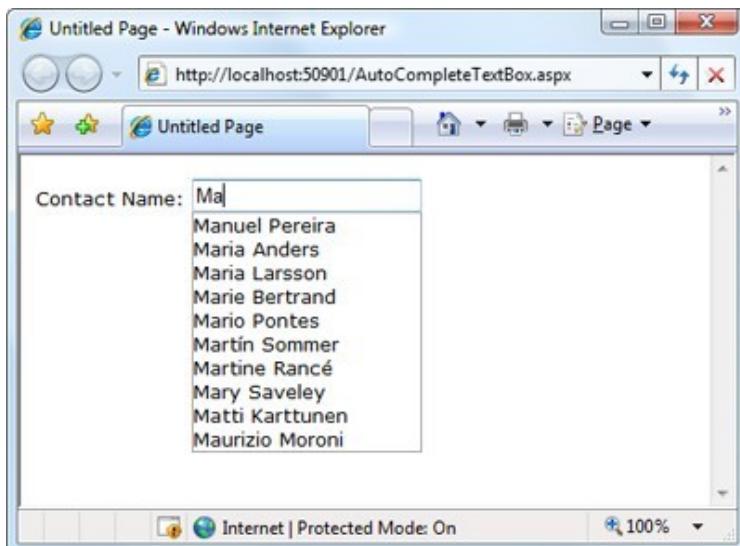


Figure 25-9. Providing an autocomplete list of names

To create this example, you need an ordinary text box, like this:

```
Contact Name:<asp:TextBox ID = "txtName" runat = "server" > </asp:TextBox>
```

Next you need to add the ScriptManager and an AutoCompleteExtender control that extends the text box with the autocomplete feature. The trick is that the list of suggestions needs to be retrieved from a specialized code routine called a *web method*, which you need to create in your page.

Note You may have heard about web methods and web services, which are remotely callable code routines that can share data between different organizations, programming platforms, and operating systems. The web method you'll use with ASP.NET AJAX isn't quite as ambitious. Although it uses some of the same plumbing, it has a simpler goal. It's really just a way for the text box to get a list of word suggestions without going through the full page life cycle.

Here's an example of how you might define the AutoCompleteExtender. It uses the TargetControlID property to bind itself to the txtName text box, and it sets the MinimumPrefixLength property to 2, which means autocomplete suggestions won't be provided until the user has entered at least two characters of text. Finally, the ServiceMethod property indicates that the web method you're going to use is named GetNames(). Before you can run this page, you need to create that method.

```
<asp:AutoCompleteExtender ID = "autoComplete1" runat = "server"
    TargetControlID = "txtName" ServiceMethod = "GetNames" MinimumPrefixLength = "2">
</asp:AutoCompleteExtender>
```

The next step is to create the GetNames() web method. Here's the basic method you need to add to the code-behind class of your web page:

```
[System.Web.Services.WebMethod]
[System.Web.Script.Services.ScriptMethod]
public static List < string > GetNames(string prefixText, int count)
{ ... }
```

The web method accepts two parameters, which indicate the text the user has typed so far and the desired number of matches (which is ten by default). It returns the list of suggestions. The two attributes that precede the GetNames() method indicate that it's a web method (meaning the client should be allowed to call it directly with HTTP requests) and that it supports JavaScript calls (which is what the AutoCompleteExtender uses).

Actually writing the code that retrieves or generates the suggestion list can be quite tedious. In this example, the code retrieves the list of name suggestions from the Northwind database. To ensure that this step is performed just once (rather than every single time the user hits a key in the text box), the name list is cached using the techniques you learned about in Chapter 23:

```
List < string > names = null;
// Check if the list is in the cache.
if (HttpContext.Current.Cache["NameList"] == null)
{
    // If not, regenerate the list. The ADO.NET code for this part
    // isn't shown (but you can see it in the downloadable examples
    // for this chapter).
    names = GetNameListFromDB();
```

```

// Store the name list in the cache for sixty minutes.
HttpContext.Current.Cache.Insert("NameList", names, null,
    DateTime.Now.AddMinutes(60), TimeSpan.Zero);
}
else
{
    // Get the name list out of the cache.
    names = (List < string >)HttpContext.Current.Cache["NameList"];
}
...

```

With the list in hand, the next step is to cut down the list so it provides only the ten closest suggestions. In this example, the list is already sorted. This means you simply need to find the starting position—the first match that starts with the same letters as the prefix text. Here's the code that finds the first match:

```

...
int index = -1;
for (int i = 0; i < names.Count; i++)
{
    // Check if this is a suitable match.
    if (names[i].StartsWith(prefixText))
    {
        index = i;
        break;
    }
}

// Give up if there isn't a match.
if (index == -1) return new List < string > ();
...

```

The search code then begins at the index number position and moves through the list in an attempt to get ten matches. However, if it reaches the end of the list or finds a value that doesn't match the prefix, the search stops.

```

...
List < string > wordList = new List < string > ();
for (int i = index; i < (index + count); i++)
{
    // Stop if the end of the list is reached.
    if (i >= names.Count) break;

    // Stop if the names stop matching.
    if (!names[i].StartsWith(prefixText)) break;
    wordList.Add(names[i]);
}
...

```

Finally, all the matches that were found are returned:

```

...
return wordList;

```

You now have all the code you need to create the effect shown in Figure 25-9.

Getting More Controls

The Accordion and AutoCompleteExtender only scratch the surface of the ASP.NET AJAX Control Toolkit, which currently includes about 50 components. The easiest way to start experimenting with other controls is to surf to www.asp.net/ajaxlibrary/act_tutorials.ashx, where you'll find a reference that describes each control and lets you try it online.

Table 25-1 highlights some of the more interesting control extenders that are available in the ASP.NET AJAX Control Toolkit, and Table 25-2 lists the controls.

Table 25-1. Control Extenders in the ASP.NET AJAX Control Toolkit

Name	Description
AlwaysVisibleControlExtender	This extender keeps a control fixed in a specific position (such as the top-left corner of the web page) even as you scroll through the content in a page.
AnimationExtender	This powerful and remarkably flexible extender allows you to add animated effects such as resizing, moving, fading, color changing, and many more, on their own or in combination.
AutoCompleteExtender	This extender allows you to supply a list of suggested entries based on partial user input. The list of entries is generated by a web service method, as described in the previous section.
BalloonPopupExtender	The extender shows a floating pop-up with the content you want when a control is clicked, moused over, or gets focus.
CalendarExtender	This extender shows a pop-up calendar that can be attached to a text box for easier entry of dates. When the user chooses a date, it's inserted in the linked control.
CascadingDropDown	This extender lets you link drop-down lists without coding the solution by hand.
CollapsiblePanelExtender	This extender lets you collapse and expand panels on your page. The rest of the page content reflows around them automatically.
ColorPickerExtender	This extender shows a pop-up color picker that can be attached to a text box for easy color selection.
ConfirmButtonExtender	This extender intercepts button clicks on a Button, LinkButton, or ImageButton control and displays a confirmation message. The Click event is suppressed if the user chooses to cancel the operation in the confirmation dialog box.
DragPanelExtender	This extender allows you to drag a panel around the page.
DropDownExtender	This extender turns almost any control into a drop-down list when the mouse moves over it. For example, you can use the DropDownExtender in conjunction with an ordinary Label control. The user can then move the mouse over it, pick a different item from the list (thereby changing the text in the label), and then move the mouse away (allowing the label to revert to its ordinary appearance).

(continued)

Table 25-1. (continued)

Name	Description
DropShadowExtender	This extender adds a graphical drop shadow effect around a panel. The drop shadow can be partially transparent, and you can control the size and roundness of its corners.
DynamicPopulateExtender	This simple extender replaces the contents of a control with the result of a web service method call.
FilteredTextBoxExtender	This extender allows you to restrict certain characters from being entered in a text box (such as letters in a text box that contains numeric data). This is meant to supplement validation, not replace it, as malicious users could circumvent the filtering by tampering with the rendered page or disabling JavaScript in the browser.
HtmlEditorExtender	This remarkably powerful extender turns an ordinary text box in a full HTML editor, complete with a toolbar of commands. The user can switch between a source view and a formatted preview. (This extender replaces the Editor control, which performed a similar function and is still included in the toolkit.)
HoverMenuExtender	This extender allows content to pop up next to a control when the user hovers over it.
ListSearchExtender	This extender allows the user to search for items in a ListBox or DropDownList by typing the first few letters of the item text. The control searches the items and jumps to the first match as the user types.
MaskedEditExtender	This extender restricts the kind of input that can be entered in a text box by using a mask. (A <i>mask</i> is a string that defines a pattern for fixed-length text, and supplies prompt characters to help the user enter the value. For example, a phone number mask might display <u>(<u>)</u> <u>-</u> <u><u></u></u> in the text box. As the user types, the placeholders are replaced with the valid numeric characters, and nonnumeric characters are rejected.) You can use the MaskedEditExtender in conjunction with the MaskedEditValidator to make sure that the user can't circumvent the JavaScript code and enter an invalid value.</u>
ModalPopupExtender	This extender allows you to create the illusion of a modal dialog box by darkening the page, disabling controls, and showing a superimposed panel on top.
MultiHandleSliderExtender	This extender transforms an ordinary TextBox control into a slider. However, this slider has features beyond the standard ASP.NET server-side equivalent. Most notably, it supports multiple thumbs that can be dragged independently along the slider track. This is useful when using a slider to indicate a range of values.
MutuallyExclusiveCheckBoxExtender	This extender allows you to associate a "key" with multiple CheckBox controls. When the user clicks a check box that's extended in this way, any other check box with the same key will be deselected automatically.

(continued)

Table 25-1. (continued)

Name	Description
NumericUpDownExtender	This extender attaches to a text box to provide configurable up- and down-arrow buttons (at the right side). These buttons increment the numeric or date value in the text box.
PagingBulletedListExtender	This extender attaches to a BulletedList and gives it client-side paging capabilities so that it can split a long list into smaller sections.
PasswordStrength	This extender attaches to a text box. As you type, it ranks the cryptographic strength of the text box value (the higher the ranking, the more difficult the password is to crack). It's meant to be used as a guideline for a password-creation box.
PopupControlExtender	This extender provides pop-up content that can be displayed next to any control.
ResizableControlExtender	This extender allows the user to resize a control with a configurable handle that appears in the bottom-right corner.
RoundedCornersExtender	This extender rounds the corners of any control for a clean, professional look.
SliderExtender	This extender converts a text box into a graphical slider that allows the user to choose a numeric value by dragging a thumb to a position on a track.
SlideShowExtender	This extender attaches to an image and causes it to display a sequence of images. The images are supplied using a web service method, and the slide show can loop endlessly or use play, pause, previous, and next buttons that you create.
TextBoxWatermarkExtender	This extender allows you to automatically change the background color and supply specific text when a TextBox control is empty. For example, your text box might include the text <i>Enter Value</i> in light gray writing on a pale blue background. This text disappears while the cursor is positioned in the text box or after you've entered a value.
ToggleButtonExtender	This extender turns the ordinary ASP.NET CheckBox into an image check box.
UpdatePanelAnimationExtender	This extender allows you to use the same animations as the AnimationExtender. However, it's designed to work with an UpdatePanel and perform these animations automatically when an update is in progress or after the panel has been refreshed.
ValidatorCalloutExtender	This extender extends the client-side logic of the ASP.NET validation controls so that they use pop-up validation callouts that point to the control with the invalid input.

Table 25-2. Controls in the ASP.NET AJAX Control Toolkit

Name	Description
Accordion	This control stacks several content panels and allows you to view one at a time. When you click a panel, that panel is expanded and the other panels are collapsed (so that just the header is visible). Additional features include an automatic fading effect and an option to limit the size of the control (in which case large content regions use scrolling when visible).
AjaxFileUpload	This control is similar to the FileUpload web control, except it does its work asynchronously, allowing the user to continue interacting with the page, and it allows the user to upload multiple files at once. On newer browsers, the AjaxFileUpload control shows the progress of each upload, and provides a drag-and-drop feature. (This control is a more powerful replacement to the AsyncFileUpload control, which is still included with the toolkit.)
ComboBox	This control is inspired by the Windows combo box, which provides a drop-down list (like the ASP.NET DropDownList control) <i>and</i> allows the user to type in arbitrary text (like the ASP.NET TextBox control). It adopts much of the programming interface and most of the conventions of its Windows counterpart.
Gravatar	This control shows a small picture that's associated with an e-mail account through the free Gravatar (globally unique avatar) service. Typically, you'd use this to show the gravatar that corresponds to a specific user (say, next to a comment left by that user). To get more information about gravatars, visit http://gravatar.com .
NoBot	This control performs several checks to attempt to detect whether an automated program (a bot) is accessing the page rather than a human. If NoBot determines that a bot is accessing the page, the request will be denied. This technique is used to prevent programs that steal content or submit comment spam to blog postings, although it can obviously be circumvented. For example, NoBot forces the browser to perform a JavaScript calculation that uses the HTML DOM and submit the result, which aims to catch a nonbrowser application accessing the page. NoBot can also reject requests that post the form back extremely quickly, or post it back a large number of times in a specific time interval. Both behaviors suggest that an automated program is at work rather than a human.
Rating	This control allows users to set a rating by moving the mouse over a series of stars until the desired number of stars are highlighted.
ReorderList	This control creates a scrollable template list that allows the user to rearrange the order of items by dragging and dropping them.
Seadragon	This advanced control allows the user to zoom in to a massively magnifiable image. Behind the scenes, these images are composed of a series of much smaller tiles, which are managed on the web server. To create a Seadragon image, you need the free Deep Zoom Composer tool (find the latest version by searching at www.microsoft.com/en-us/download). It's also worth noting that these zoomable images are a much hyped feature that's built into Silverlight.
TabContainer	This control resembles the tabs shown in a Windows application. Each tab has a header, and the user moves from one tab to another by clicking the header.
Twitter	Shows a panel with a list of tweets (messages from the Twitter service). You can show the most recent tweets sent by a particular Twitter user, or the most recent tweets that match a search expression.

The Last Word

The Ajax support that you'll find in ASP.NET AJAX is surprisingly powerful—and practical. As you saw in this chapter, the UpdatePanel, UpdateProgress, and Timer controls give you a nearly painless way to take a traditional postback-heavy web form and give it a more responsive Ajax-powered user interface, often without requiring new code. And if you want to step further into the world of Ajax frills and get collapsible panels, autocompleting text boxes, and a wide range of animated effects, you need to look no further than the ASP.NET AJAX Control Toolkit, which provides a lot more to play with.

To learn more about ASP.NET AJAX, check out the Microsoft ASP.NET AJAX site at www.asp.net/ajax and the ASP.NET AJAX Control Toolkit at www.asp.net/ajaxlibrary/act.ashx.