UNIT - 1
THE .NET FRAMEWORK

Chapter 1: The .NET Framework

Overview

Once again, Microsoft is doing what it does best: creating innovative new technologies and wrapping them in marketing terms that cause widespread confusion. Most developers have finally started to sort out buzzwords like ActiveX and Windows DNA, and are now faced with a whole new set of jargon revolving around .NET and the CLR. So exactly what does it all mean?

This chapter examines the technologies that underlie .NET. I'll present a high-level overview that gives you your first taste of some of the radical changes that will affect ASP.NET applications, and some of the long-awaited improvements that are finally in place. I'll also explain the philosophy that led to the creation of the .NET platform, and show how it fits into the wider world of development—and Microsoft's long-term plans.

The .NET Programming Framework

The starting point for any analysis of the .NET framework is the understanding that .NET is really a cluster of different technologies. It includes:

- **The .NET languages**, which include C# and Visual Basic .NET, the object-oriented and modernized successor to Visual Basic 6.0.
- **The Common Language Runtime (CLR)**, the .NET runtime engine that executes all .NET programs, and provides modern services such as automatic memory management, security, optimization, and garbage collection.
- **The .NET class library**, which collects thousands of pieces of prebuilt functionality that you can snap in to your applications. These are sometimes organized into technology sets, such as ADO.NET (the technology for creating database applications) and Windows Forms (the technology for creating desktop user interfaces).
- **ASP.NET**, the platform services that allow you to program web applications and Web Services in any .NET language, with almost any feature from the .NET class library.
- **Visual Studio .NET**, an optional development tool that contains a rich set of productivity and debugging features.

The division between these components is not sharp. For example, ASP.NET is sometimes used in a very narrow sense to refer to the portion of the .NET class library used to design web forms. On the other hand, ASP.NET is also used to refer to the whole topic of .NET web applications, which includes language and editor issues, and many fundamental pieces of the class library that are not web-specific. That's generally the way that the term is used in this book. Our exhaustive examination of ASP.NET includes .NET basics, the VB .NET language, and topics that any .NET developer could use, such as component-based programming and ADO.NET data access.
The .NET class library and Common Language Runtime—the two fundamental parts of .NET—are shown in Figure 1-1.

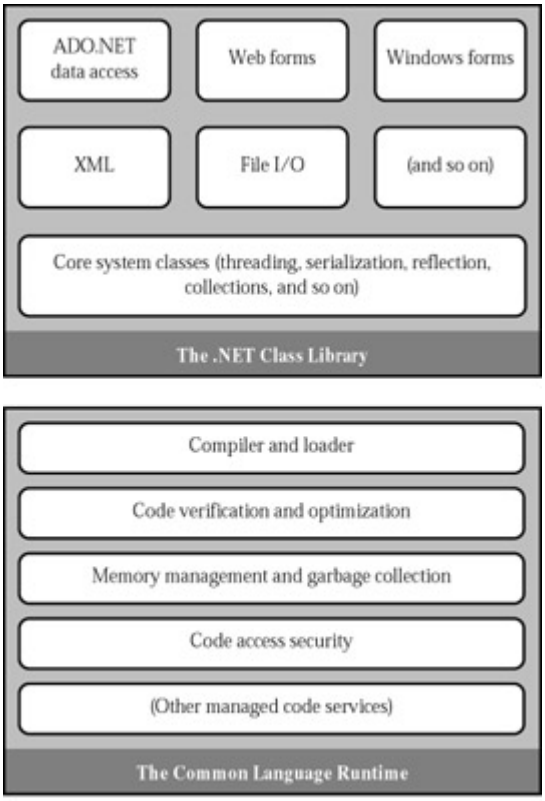


Figure 1-1: The .NET framework

VB .NET, C#, and the .NET Languages

Visual Basic .NET (sometimes called VB .NET) is a redesigned language that improves on traditional Visual Basic, and even breaks compatibility with existing VB programs. Migrating to Visual Basic .NET is a stretch—and a process of discovery for most seasoned VB developers. The change is even more dramatic if you are an ASP developer used to the scaled-down capabilities of VBScript.

C#, on the other hand, is an entirely new language. It resembles Java and C++ in syntax, but there is no direct migration path. C++ programmers will find that many of the frustrating tangles have been removed from their language in C#, while Java programmers will find that C# introduces a host of new features and coding enhancements. In short, C#, like Visual Basic .NET, is an elegant, modern language ideal for creating the next generation of business applications.

Interestingly, C# and Visual Basic .NET are actually far more similar than Java and C# or Visual Basic 6 and VB .NET. Though the syntax is different, both use the .NET class library and are supported by the Common Language Runtime. In fact, almost any block of C# code can be translated, line by line, into an equivalent block of VB .NET code. There is the occasional language difference (for example, C# supports operator overloading while VB .NET does not), but for the most part, a developer who has learned one .NET language can move quickly and efficiently to another.

In fact, all the .NET languages are compiled to the same intermediate language, which is known as MSIL, or just IL. The CLR only runs IL code, which is the reason why the C# and VB .NET languages are so similar (and perform essentially the same). Figure 1-2 shows the compilation process, with a bit of simplification—in an ASP.NET application, a final machine-specific executable is cached on the web server to provide the best possible performance.

from VBScript by two major evolutionary leaps. All of these features are available in other .NET languages such as C#, but Visual Basic and VBScript developers will have to deal with some of the greatest changes from what they know.

Learning About Visual Basic .NET and Objects

These dramatic changes present some challenges to even the most experienced of developers. In the next two chapters, we'll sort through the new syntax of Visual Basic .NET, and I'll introduce the basics of object-oriented programming. By learning the fundamentals before you start creating simple web pages, you'll face less confusion and move more rapidly to advanced topics like database access and web services.

The Common Language Runtime

The Common Language Runtime (CLR) is the engine that supports all the .NET languages. Most modern languages use runtimes, such as msvbvm60.dll for Visual Basic 6 applications and mscrt40.dll for C++ applications. These runtimes may provide libraries used by the language, or they may have the additional responsibility of executing the code (as with Java).

Runtimes are nothing new, but the CLR represents a radical departure from Microsoft's previous strategy. To start, the CLR and .NET framework are much larger and more ambitious. The CLR also provides a whole set of related services such as code verification, optimization, and garbage collection, and can run the code from any .NET language.

The CLR is the reason that some developers have accused .NET of being a Java clone. As with Java, .NET applications run inside a special managed environment with garbage collection and an intermediate language, and acquire features from a rich class library. (The claim is fairly silly. It's true that .NET is quite similar to Java in key respects, but it's also true that every programming language "steals" and improves from previous programming languages. This includes Java, which adopted parts of the C/C++ language and syntax. Of course, in many other aspects .NET differs just as radically from Java as it does from VBScript code.)

The implications of the CLR are wide-ranging:

Deep language integration C# and VB .NET, like all .NET languages, compile to a special Intermediate Language called MSIL (or just 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 just language compatibility; it's really language integration.

No more "DLL hell" IL programs store extra information about their classes and the components they require (called metadata). The CLR examines this information and automatically prevents an application from using the wrong version of a component.

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 will be 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 monitors memory, automatically removes objects that aren't being used with its garbage collection, and prevents the wide variety of memory mistakes that are possible with pointers and C++. For Visual Basic programmers, many of these advantages are nothing new—they are similar to features of the traditional VB framework.

Along with these truly revolutionary benefits, there are some other potential drawbacks, and some issues that haven't yet been answered:

Performance A typical ASP application will gain considerably in performance because of various improvements, including automatic caching and precompiling. However, .NET components or other applications probably won't match the blinding speed of well-written C++, because the CLR imposes many other safety checks and additional overhead. Generally, this will be a factor only in a few performance-critical high-workload applications (like real-time games). With high-volume web applications, the potential bottlenecks are rarely processor-related, but usually tied to the speed of an external resource such as a database or the server's file system. ASP.NET caching and well-written database code can ensure excellent performance for a web application.

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

Cross-platform? No one is entirely sure whether .NET is destined for use on other operating systems and platforms. Currently, .NET programs will make the switch to upcoming 64-bit versions of Windows without a problem. Other rumors persist, including suggestions that we will see an ASP.NET service that runs on Apache, or a limited .NET framework that allows users to develop Unix applications. This is just speculation, however, and .NET will probably never have the wide reach of a language like Java because it incorporates too many different platform-specific and operating-system – specific technologies and features.

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 email 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 its features, by interacting with a consistent set of objects. This helps encourage consistency among different .NET languages, and removes the need to install numerous separate components and SDKs.

Some parts of the class library include features you'll never need to use (such as the classes used to create desktop applications with the Windows interface). Other parts of the class library are targeted directly at web development, enabling Web Services, web form user interface, and countless other utility classes. Still more classes can be used in a number of different 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 file I/O, data access, and XML information, to name just a few.

Other characteristics of the .NET framework include:

Open standards Microsoft currently provides programming tools that allow you to

work with many open standards, such as XML and SOAP. In .NET, however, many of these standards are baked in to the framework. For example, ADO.NET (Microsoft's data access technology) uses XML natively, behind the scenes. Similarly, Web Services work automatically through XML and HTTP. This deep integration of open standards makes cross-platform work much easier.

Disconnected model The .NET framework inherits many of the principles of Microsoft's DNA programming model. This includes an emphasis on distributed and Internet applications. Technologies like ADO.NET are designed from the ground up for disconnected, scalable access, and don't need any special code to be used in an ASP.NET application.

Emphasis on infrastructure Microsoft's philosophy is that they will provide the tedious infrastructure so that application developers only need to write business-specific code. For example, the .NET framework automatically handles files, message queues, databases, and remote method calls (through Web Services). You just add the logic needed for your organization.

ASP.NET

ASP.NET is a part of the .NET framework. As a programmer, you interact with it by using the appropriate types in the class library to write programs and design web forms. When a client requests a page, the ASP.NET service runs (inside the CLR environment), executes your code, and creates a final HTML page to send to the client.

To understand ASP.NET's features, it helps to understand ASP's limitations. In other words, before you can understand the .NET solution, you need to understand the problems developers are struggling with today.

Scripting limitations ASP applications rely on VBScript, which suffers from a number of limitations. To overcome these problems, developers usually need to add separately developed components, which add a new layer of complexity. In ASP.NET, web pages are designed in a modern .NET language, not a scripting language.

Headaches with deployment and configuration Because of the way COM and ASP work, you can't easily update the components your web site uses. Often, you need to manually stop and restart the server, which just isn't practical on a live web server. Changing configuration options can be just as ugly. ASP.NET introduces a slew of new features to allow web sites to be dynamically updated and reconfigured.

No application structure ASP code is inserted directly into a web page along with HTML markup. The resulting tangle has nothing in common with today's modern, object-oriented languages. As a result, web form code can rarely be reused or modified without hours of effort.

State limitations One of ASP's strongest features is its integrated session state facility. However, session state is useless in scenarios where a web site is hosted by several separate web servers. In this scenario, a client might access server B while its session information is trapped on server A and essentially abandoned. ASP.NET corrects this problem by allowing state to be stored in a central repository: either a separate process or a database that all servers can access.

Visual Studio .NET

The last part of .NET is the optional Visual Studio .NET editor, which provides a rich environment

where you can rapidly create advanced applications. Some of its features include:

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

Debugging tools Visual Studio .NET retains its legendary debugging tools that allow you to watch your code in action and track the contents of variables.

Page design You can create an attractive page with drag-and-drop ease using Visual Studio .NET's integrated web form designer.

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

Does this Book Use Visual Studio .NET?

I recommend Visual Studio .NET highly. However, in this book you will start with the basics of ASP.NET, code behind, and web control development using Notepad. This is the best way to get a real sense of what is happening behind the scenes. Once you understand how ASP.NET works, you can quickly master Visual Studio .NET, which I introduce in Chapter 8. I hope you'll use Visual Studio .NET for all the remaining examples, but if you prefer Notepad or a third-party tool, you can use that instead—with a little more typing required.

Chapter 2: Learning the .NET Languages

Overview

In order to create an ASP.NET application, you need to choose a .NET language to program it. If you're an ASP or Visual Basic developer, the natural choice is VB .NET. If you are a long-time Java programmer or old-hand C++ coder, C# will probably suit you best. VBScript isn't an option—as discussed in Chapter 1, it just doesn't have the range, flexibility, and elegance a modern language demands.

In this chapter, you'll learn about the basic features of all .NET languages. This includes the common set of data types, types of variable operations, and use of functions, events, enumerations, and attributes. These ingredients are built into the Common Language Runtime. They are available (and work almost exactly the same) in any .NET language.

While you learn about these elements, you'll also learn the corresponding VB .NET or C# syntax you need to use them. This syntax is really just the top layer of .NET, which sits like a thin skin over a common engine (the CLR). The more you learn about .NET, the more you'll realize that all .NET languages share the same concepts and require the same understanding.

Data Types

Every .NET language uses the same variable data types. Different languages may provide slightly different names (for example, a VB .NET Integer is the same as a C# int), but the CLR makes no distinction. This allows the deep language integration that was discussed in Chapter 1. Because languages share the same data types, you can easily use classes from one .NET language in a client written in another .NET language. Variables can be exchanged natively, and no conversion is required.

In order to create this system, Microsoft needed to iron out many of the inconsistencies that existed between VBScript, Visual Basic 6, C++, and other languages. Their solution was to create a set of basic data types, which are provided in the .NET class library. These data types are shown in Table 2-1.

Table 2-1: Common Data Types

Class Library Name	VB .NET Name	C# Name	Contains
Byte	Byte	byte	An integer from 0 to 255.
Int16	Short	short	An integer from –32768 to 32767.
Int32	Integer	int	An integer from –2,147,483,648 to 2,147,483,647.
Int64	Long	long	An integer from about –9.2e18 to 9.2e18.
Single	Single	float	A single-precision floating point number from approximately –3.4e38 to 3.4e38.
Double	Double	double	A double-precision floating point number from approximately –1.8e308 to 1.8e308.
Decimal	Decimal	decimal	A 128-bit fixed-point fractional number that supports up to 28 significant digits.
Char	Char	char	A single 16-bit Unicode character.
String	String	string	A variable-length series of Unicode

			characters.
Boolean	Boolean	bool	A True or False value (or lowercase true or false in C#).
DateTime	Date	N/A[*]	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, year 9999. Time values can resolve values to 100 nanosecond increments. Internally, this data type is stored as a 64-bit integer.
TimeSpan	N/A[*]	N/A[*]	Represents a period of time, as in 10 seconds or 3 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.
[*]If the language does not provide an alias for a given type, you can use the .NET class name.			

Data Type Changes from Visual Basic 6

Luckily for VB developers, the list of data types in .NET is quite similar to the data types from earlier versions. Some changes, however, were unavoidable:

- Visual Basic defined an Integer as a 16-bit number with a maximum value of 32,767. VB .NET, on the other hand, defines an Integer as a 32-bit number. Long now refers to a 64-bit integer, while Short refers to a 16-bit integer.
- The Currency data type has been replaced by Decimal, which supports more decimal places (and thus greater accuracy).
- Fixed-length strings are no longer supported.
- Variants are no longer supported and have been replaced with the generic Object type, which can hold any object or data type.
- The Date data type has increased range and precision and works slightly differently. Depending on your code, you may need to rewrite date manipulation code.

Declaring Variables

To create a variable, you need to identify a data type and a variable name. In VB .NET, the name comes first, after a special Dim keyword. In C#, the data type starts the line.

VB .NET

```
Dim ErrorCode As Integer
Dim MyName As String
```

C#

```
int errorCode;
string myName;
```

In C#, the convention is to begin variable names with a lowercase letter, unless you are creating a public variable that can be seen from other classes. Public variables or properties should always use an initial capital.

You can also create a variable by using the type from the .NET class library. This approach produces identical variables. It's also a requirement if you need to create a variable to hold an object or a type of data that doesn't have an alias built into the language.

VB .NET

```
Dim ErrorCode As System.Int32
Dim MyName As System.String
```

C#

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

These examples use fully qualified type names that indicate that the Int32 type is found in the System namespace (along with all the fundamental types). In Chapter 3, we'll discuss types and namespaces in more detail.

What's in a Name: Data Types

You'll notice that the preceding examples don't use variable prefixes. Most Visual Basic and C++ programmers are in the habit of adding a few characters to the start of a variable name to indicate its data type. In .NET, this practice is de-emphasized because data types can be used in more flexible ways without any problem, and most variables hold references to full objects anyway. In this book, variable prefixes are not used, except for web controls, where it helps to distinguish lists, text boxes, buttons, and other common user interface elements. In your own programs, you should follow a consistent (typically company-wide) standard that may or may not adopt a system of variable prefixes.

Initializers

You can also assign a value to a variable in the same line that you create it. This feature is common to C++ programmers, but a new addition to VB .NET.

VB .NET

```
Dim ErrorCode As Integer = 10
Dim MyName As String = "Matthew"
```

C#

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

In Visual Basic, you can use a simple data variable without initializing it. Numbers will be automatically initialized to 0, and strings to an empty string (""). In C#, however, you must assign a value to a variable before you attempt to read from it. For example, the following statement will succeed in VB .NET, but would fail in C# if the Number variable had not been initialized to a value.

```
Number = Number + 1
```

Arrays

Visual Basic programmers will find that arrays are one of the most profoundly changed language elements. In order to harmonize .NET languages, Microsoft decided that all arrays must start at a fixed lower bound of 0. There are no exceptions.

Arrays still have a few subtle differences in C# and in VB .NET, though. For example, when you create an array in VB .NET, you specify the upper bound, while in C# you specify the number of elements. This change is bound to go unnoticed by many programmers until it leads to inevitable errors.

VB .NET

```
' Create an array with 10 strings (from index 0 to index 9).
Dim StringArray(9) As String
```

C#

```
// Create an array with 9 strings (from index 0 to index 8).
// Unlike in VB .NET, you need to initialize the array with empty
// values in order to use it.
string[] stringArray = new string[9];
```

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.

VB .NET

```
Dim MyArray() As String = {"1", "2", "3", "4"}
```

C#

```
string[] myArray = {"1", "2", "3", "4"};
```

The same technique works for multi-dimensional arrays, except two sets of curly brackets are required:

VB .NET

```
Dim MyArray() As Integer = {{1, 1}, {2, 2}, {3, 3}, {4, 4}}
```

C#

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

To access an array, you specify the index number. In Visual Basic, you use ordinary brackets, while in C# you use square brackets.

VB .NET

```
Dim Element As Integer
Element = MyArray(1, 2)
```

C#

```
int element;
element = myArray[1, 2];
```

Arrays can include any data type, including objects. They also automatically support read-only iteration. This means you can use a For Each syntax to loop through all the elements in an array. This is a common trick that is often used with collections, but .NET allows you to use it with arrays as well.

VB .NET

```
Dim StringArray() As String = {"one", "two", "three"}
Dim Element As String

For Each Element In MyStringArray
    ' This code loops three times, with the Element variable set to
    ' "one", then "two", and then "three".
Next
```

C#

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

// Note that you don't need to define the element variable first in C#.
// Instead, you define it "inside" the foreach brackets.
foreach (string element in myStringArray)
{
    // This code loops three times, with the element variable set to
    // "one", then "two", and then "three".
}
```

In a multi-dimensional array, the For Each iteration would move over each column in a single row first, and then move to the next row.

One nice feature that VB .NET offers but C# lacks is array redimensioning. In VB .NET, all arrays start with an initial size, and any array can be resized. To resize an array, you use the ReDim keyword.

```
Dim MyArray(10, 10) As Integer
ReDim MyArray(20, 20)
```

In this example, all the contents in the array will be erased when it is resized. To preserve the contents, you can use the optional Preserve keyword when redimensioning the array. However, if you are using a multi-dimensional array you will only be able to change the last dimension, or a runtime error will occur.

```
Dim MyArray(10, 10) As Integer
'Allowed, and the contents will remain.
ReDim Preserve MyArray(10, 20)
' Not allowed. A runtime error will occur.
ReDim Preserve MyArray(20, 20)
```

To achieve the same effect in C#, you need to write manual array copying code, or just use a collection class such as the ArrayList, which always allows a dynamic size.

Collections Replace Arrays in Many Scenarios

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

Enumerations

An enumeration is group of related constants. In reality, every enumerated value just corresponds to a preset integer. In your program, however, you refer to an enumerated value by name, which makes your code clearer and helps to prevent errors.

Enumerations cannot be declared inside a procedure. Instead, they are declared at the class or module level using a special block structure.

VB .NET

```
' Define an enumeration called UserType with three possible values.
Enum UserType
   Admin
   Guest
   Invalid
End Enum
```

C#

```
// Define an enumeration called UserType with three possible values.
enum UserType
{
   Admin,
   Guest,
   Invalid
}
```

Now you can use the UserType enumeration as a special data type that is restricted to one of three possible values. You set or compare the enumerated value using a special dot syntax.

VB .NET

```
Dim NewUserType As UserType = UserType.Admin
```

C#

```
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 Invalid. You can set a number directly into an enumeration, although this can lead to an undetected error if you use a number that doesn't correspond to a named value.

In some cases, however, you want to control what numbers are set for various enumerations. This technique is typically used when the number corresponds to something else (for example, an error code returned by a legacy piece of code).

VB .NET

```
Enum ErrorCode
   NoResponse = 166
   TooBusy = 167
   Pass = 0
End Enum
```

C#

```
enum ErrorCode
{
   NoResponse = 166,
   TooBusy = 167,
   Pass = 0
}
```

You could then use this ErrorCode enumeration with a function that returns an integer.

VB .NET

```
Dim Err As ErrorCode
Err = DoSomething()
If Err = ErrorCode.Pass
   ' Operation succeeded.
End If
```

C#

```
ErrorCode err = DoSomething()
if (err == ErrorCode.Pass)
{
   // Operation succeeded.
}
```

Enumerations Are Widely Used in .NET

You may not need to create your own enumerations to use in ASP.NET applications, unless you are designing your own components. However, the concept of enumerated values is extremely important, because the .NET class library uses it extensively. For example, you set colors, border styles, alignment, and various other web control styles using enumerations provided in the .NET class library.

Scope and Accessibility

The variable declaration examples shown so far are ideal for creating variables inside a code procedure (for example, a variable for a temporary counter). You can also create a variable outside a procedure that belongs to an entire class or module. In Visual Basic 6, this was often known as a form-level variable.

In this case, you can still create the variable in the same way, or you can explicitly set the access level by adding a keyword.

VB .NET

```
' A public class-level or module-level variable.
Public ErrorCode As Integer
```

C#

```
// A public class-level variable.
public int errorCode;
```

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 2-2 explains the different access levels you can use.

Table 2-2: Accessibility Keywords

VB .NET Keyword	C# Keyword	Accessibility
Public	public	Can be accessed by any other class.
Private	private	Can only be accessed by code procedures inside the current class. This is the default if you don't specify an accessibility keyword.
Friend	internal	Can be accessed by code procedures in any of the classes in the current assembly.
Protected	protected	Can be accessed by code procedures in the current class, or any class that inherits from this class.
Protected Friend	protected internal	Can be accessed by code procedures in the current application, or any class that inherits from this class.

Accessibility is defined on a class-by-class basis. For the purposes of this discussion, a class is just a container for related code routines and variables. Chapter 3 will go into much more detail. If you are new to class-based programming, don't worry about understanding all of these different options. The full list is included for completeness only, or as a reference when you start to design your own custom classes and components.

You can use initializers with class variables in the same way that you use them with procedure-level variables. These variables will be automatically created and initialized when the class is first created.

Scope Changes from Visual Basic 6

In traditional Visual Basic, there were only two types of scope for a variable: form level and procedure level. VB .NET tightens these rules a notch by introducing block scope. Block scope means that any variables created inside a block structure (such as a conditional If ⋯ End If or a For ⋯ Next or a Do ⋯ Loop) are only accessible inside that block of code.

```
For i = 0 To 10
    Dim TempVariable As Integer
    ' (Do some calculation with TempVariable.)
Next
' You cannot access TempVariable here.
```

This change won't affect many programs. It's really designed to catch a few more accidental errors.

Variable Operations

You can use the standard type of variable operations in both languages, including math symbols (+, -, /, *, and ^ for exponents), and string concatenation (+ or & in VB .NET). In addition, Visual Basic .NET now provides special shorthand assignment operators to match C#. A few examples are shown below.

VB .NET

Table 2-6: Useful Array Members

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

Conditional Structures

Both C# and VB .NET use an equivalent syntax for evaluating conditions. The if block in C# is slightly less verbose, and uses brackets to identify the expression and curly braces to group the conditional code. The comparison operators (>, <, and =) are the same, with the exception that C# uses two equal signs to indicate comparison. This prevents it from being confused with a statement that assigns a value to a variable.

VB .NET

```
If MyNumber > 10 Then
   ' Do something.
ElseIf MyString = "hello" Then
   ' Do something.
Else
   ' Do something.
End If
```

C#

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

Both languages also provide a Select Case structure that you can use to evaluate a single variable or expression for multiple possible values. In this case, the C# syntax is slightly more awkward because it inherits the convention of C/C++ programming, which requires that every conditional block of code is ended by a special break keyword.

VB .NET

```
Select Case MyNumber
   Case 1
      ' Do something.
   Case 2
      ' Do something.
   Case Else
      ' Do something.
End Select
```

C#

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

Loop Structures

One of the most basic ingredients in many programs is the For ⋯ Next loop, which allows you to repeat a given block of code a set number of times. In Visual Basic, you specify a starting value, an ending value, and the amount to increment with each pass.

```
Dim i As Integer
For i = 1 To 10 Step 1
   ' This code executes 10 times.
Next
```

The "Step 1" clause specifies that i will be increased by 1 for each new loop. You can omit this part of the code, as 1 is the default increment.

In C#, the for loop has a similar syntax, with a few differences. First of all, you don't need to define the counter variable first; it is usually created in the actual loop definition. You'll also notice that you explicitly specify the condition that is tested to decide whether the loop should continue. In Visual Basic code, this condition is implicit—the code always checks if the counter variable has reached or exceeded the specified target.

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

As you've already seen with the array example, both languages also provide a For Each block that allows you to loop through the items in a collection.

Visual Basic also supports a Do ⋯ Loop structure that tests a specific condition using the While or Until keyword. These two keywords are opposites: While means "as long as this is true," and Until means "as long as this is not true." C# dispenses with this duplication, and only supports a while condition.

```
Dim i As Integer = 1
Do
   i += 1
   ' This code executes 10 times.
```

```
Loop While i < 10
```

You can also place the condition at the beginning of the loop. In this case, the condition is tested before the loop is started. This code is equivalent, unless the condition you are testing is False to begin with. In that case, none of the code in the loop will execute. If the condition is evaluated at the end of the loop, the code will always be executed at least once.

```
Dim i As Integer = 1
Do While i < 10
    i += 1
    ' This code executes 10 times.
Loop
```

In C#, these two types of loops (with the condition at the beginning and the condition at the end) are treated differently. The while loop, which places the condition at the beginning, is the most common.

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

The do ⋯ while block allows you to evaluate the condition at the end.

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

Sometimes you need to exit a loop in a hurry. In C#, you can use the break statement to exit any type of loop. In Visual Basic, you need to use the corresponding Exit statement (such as Exit For, or Exit Do). C# provides an additional statement that Visual Basic does not. The continue keyword automatically skips the rest of the code in the loop and continues from the start with a new iteration.

Functions and Subroutines

Visual Basic distinguished between two types of procedures: ordinary subroutines and functions, which return a value. As in past versions of Visual Basic, you can set the return value by assigning a value to the function name. VB .NET also introduces the useful Return keyword, which allows you to exit a function and return a result in one quick, clear step.

```
Private Sub MySub()
    ' Code goes here.
End Sub

Private Function MyFunc() As Integer
    ' As an example, return the number 10.
    Return 10
End Function
```

C# uses an inverted syntax that places the return value first, followed by the procedure name. The procedure name must always be followed by parentheses, even if it does not accept parameters, in order to identify it as a procedure. Subroutines are differentiated by functions in that they specify the keyword void to indicate that no information is returned.

```
void MySub()
{
   // Code goes here.
}

int MyFunc()
{
   // As an example, return the number 10.
   return 10;
}
```

In this case, the C# examples omit the Private keyword. This is just a C# convention; you could include it if you want (or you could omit the Private keyword from the Visual Basic code). As in Visual Basic, all procedures are private by default. Alternatively, you can also specify any of the access keywords described in Table 2-2 to change the accessibility of a procedure.

Parameters

Procedures can also accept additional information through parameters. Parameters are defined in a similar way to variables. By convention, parameters always begin with a lowercase first letter in any language.

VB .NET

```
Private Function AddNumbers(number1 As Integer, number2 As Integer)
As Integer
   Return (number1 + number2)
End Sub
```

C#

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

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

VB .NET

```
' Call a subroutine with no parameters.
MySub()

' Call a subroutine with two Integer parameters.
MySub(10, 20)

' Call a function with two Integer parameters and an Integer return
value.
Dim ReturnValue As Integer = AddNumbers(10, 10)
```

C#

```
// Call a subroutine with no parameters.
MySub();

// Call a subroutine with two Integer parameters.
MySub(10, 20);

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

Parameter Changes from Visual Basic 6

Visual Basic now requires parentheses around the parameters for any procedure call, regardless of whether the procedure returns a value. This new syntax is clearer than before, when parentheses were only used when receiving a return value from a function. Visual Basic also now defaults to ByVal for all parameters, ensuring a greater level of protection and clarity. (Although for the sake of clarity, many of the code examples in this book omit the extra ByVal keywords for parameters.)

Finally, VB .NET now requires that optional parameters have a default value specified.

```
Private Sub DoSomething(Optional taskNumber As Integer = 20)
```

As a side effect, the IsMissing function is no longer supported.

By Reference and By Value Parameters

You can create two types of procedure parameters. The standard (known as ByVal in Visual Basic) ensures that any changes do not propagate back to the calling code and modify the original variable. In other words, a copy of the variable's value is submitted to the procedure. Another option is ByRef (known as ref in C#), which allows you to modify the original variable.

VB .NET

```
Private Sub ProcessNumber(ByRef number As Integer)
    number *= 2
End Sub
```

C#

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

With ByRef, the calling code must submit a variable, not a literal value. In C# code, the ref value must also be specified in the function call, which indicates that the calling code is aware the parameter value could be changed.

VB .NET

```
Dim Num As Integer = 10
ProcessNumber(Num)
```

C#

```
int num = 10;
ProcessNumber(ref num);
```

C# Also Supports an out Parameter

The out keyword works just like the ref keyword, but it allows the calling code to use an uninitialized variable as a parameter, which is otherwise forbidden in C#. This wouldn't be appropriate for the ProcessNumber procedure, because it reads the submitted parameter value (and then doubles it). If, on the other hand, the variable is not used at all by the procedure except

to return information, you can use the out keyword. In this case, C# won't try to submit the value of the parameter to the procedure, but it will return the value that was set in the procedure. As with the ref keyword, out must be specified before the parameter in the procedure declaration *and* in the procedure call.

Note that Visual Basic .NET does not require this keyword, because Visual Basic always initializes new variables.

Procedure Overloading

C# and VB .NET both support overloading, which allows you to create more than one function and subroutine with the same name, but that accepts a different list of parameters (or a similar parameter list with different data types). The CLR will automatically choose the correct procedure for a given call by examining the list of parameters the calling code uses.

This is old hat to C programmers, but a valuable new introduction to the Visual Basic world. It allows you to collect different versions of several functions together. For example, you might allow a database search that returns an author name. Rather than create three functions with different names depending on the criteria (such as GetNameFromID, GetNameFromSSN, GetNameFromBookTile), you could create three versions of the GetCustomerName function.

To overload a procedure, you use the Overloads keyword in VB .NET. No additional keyword is required in C#.

VB .NET

```
Private Overloads Function GetCustomerName(ID As Integer) As String
    ' Code here.
End Function

Private Overloads Function GetCustomerName(bookTitle As String) _
  As String
    ' Code here.
End Function

' And so on...
```

C#

```
string GetCustomerName(int ID)
{
    // Code here.
}

string GetCustomerName(string bookTitle)
{
    // Code here.
}

// And so on...
```

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

Overloaded Methods Are Used Extensively in .NET

.NET uses overloaded methods in most of its classes, allowing you to use a range of different parameters, but centralizing functionality under common names. Even the methods we've looked at so far (like the String methods for padding or replacing text) have multiple versions that provide similar features with various options. .NET classes never use optional parameters.

Delegates

Delegates are a new feature of .NET languages. They allow you to create a variable that refers to a procedure. You can use this variable at any time to invoke the procedure.

The first step when using a delegate is to define its signature. Delegate variables can only point to functions or procedures that match its specific, defined signature (list of parameters). This allows a level of type safety that isn't found with traditional function pointers in languages like C++.

VB .NET

```
' Define a delegate that can represent any function that accepts
' a single string parameter and returns a string.
Private Delegate Function StringFunction(In As String) As String
```

C#

```
// Define a delegate that can represent any function that accepts
// a single string parameter and returns a string.
delegate string StringFunction(string In);
```

Once you have defined a type of delegate, you can create and assign a delegate variable. For example, assume your program has the next function.

VB .NET

```
Private Function TransalteEnglishToFrench(english As String) As String
    ' Code goes here.
End Function
```

C#

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

This function matches the delegate definition because it requires one string and returns a string. That means you can create a variable of type StringFunction, and use it to store a reference to this function.

VB .NET

```
' Create a delegate variable.
Dim FunctionReference As StringFunction

' Store a reference to a matching procedure.
FunctionReference = AddressOf TranslateEnglishToFrench
```

C#

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

// Store a reference to a matching procedure.
functionReference = TranslateEnglishToFrench()
```

This code is equivalent, but the C# statement does not require the special AddressOf keyword. If you try the same syntax with VB .NET, your program will run the function, return a value, and attempt to apply that value to the delegate variable, which will generate an error.

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

VB .NET

```
Dim FrenchString As String
FrenchString = FunctionReference("Hello")
```

C#

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

In the code example above, the procedure that the FunctionReference delegate points to will be invoked with the parameter "Hello" and the return value will be stored in the FrenchString variable.

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? That would allow the client application to have multiple "listeners," and notify them all at once when something happens.

In fact, delegates do have this functionality, but you are more likely to see it in use with .NET events. Events, which are described in Chapter 3, are based on delegates, but work at a slightly higher level and provide a few automatic niceties. In a typical ASP.NET program, you'll use events extensively, but probably never work directly with delegates.

UNIT - 2
TYPES, OBJECT AND NAMESPACE

Chapter 3: Types, Objects, and Namespaces

Overview

Object-oriented programming has been a popular buzzword over the last several years. In fact, one of the few places that object-oriented programming *wasn't* emphasized was in ordinary ASP pages. With .NET, the story changes considerably. Not only does .NET allow you to use objects, it demands it. Almost every ingredient you'll need to use to create a web application is, on some level, really a kind of object.

So how much do you need to know about object-oriented programming to write .NET pages? It depends 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 a finer grain of control. This book assumes that if you are willing to pick up a thousand-page book, you are the latter type of programmer—one who excels by understanding how things work, and why they work the way that they do. It also assumes that you are interested in some of the advance ASP.NET programming that *will* require class-based design, like designing custom controls (see Chapter 22) and creating your own components (see Chapter 21).

This chapter explains objects from the point of view of the .NET framework. I won't rehash the typical object-oriented theory, because there are already countless excellent programming books on the subject. Instead, I'll show you the types of objects .NET allows, how they are constructed, and how they fit into the larger framework of namespaces and assemblies.

The Basics About Classes

As a developer, you've probably already created classes, or at least heard about them. Classes are the 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 tha 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. These instances are called objects.

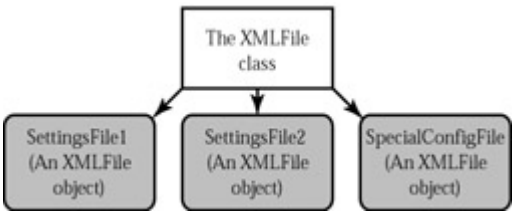


Figure 3-1: Classes make objects

Every class is made up of exactly three things:

Properties These store information about an object. Some properties may be read-only (so they cannot be modified), while others can be changed by the code that is using the object. For example, ASP pages provide a built-in Request object that you can use to retrieve information about the current client request. One property of this object, Cookies, provides a collection of cookies that were sent with the request.

Methods These allow you to perform an action with an object. (You can also interact with an object by setting properties, but methods are used for compound actions that perform a distinct task or may change the object's state significantly.) For example, ASP pages provide a built-in Server object that provides a CreateObject method you can use to run a COM component on the server.

Events These provide notification from the control that something has happened. In traditional ASP programming, you saw events only in the global.asa file, and they were not clearly tied to any specific object. If you've programmed ordinary Visual Basic programs, you have seen how controls fire events to notify your code about mouse clicks and data changes. ASP.NET controls provide a similar (although reduced) set of events.

In addition, classes contain their own code and internal set of private data. Classes behave like black boxes, which means that when you use one, you shouldn't waste any time wondering about what low-level information it's retaining. When using a class, you only need to worry about its public interface, which is the set of properties, methods, and events you have to work with. Together, these elements are called class members.

In ASP.NET, you will create your own custom classes to represent individual web pages. In addition, you will create custom classes if you design any special components (as you'll see in Chapter 21). For the most part, however, you'll be using prebuilt classes from the .NET class library, rather than programming your own.

Shared Members

One of the tricks about .NET classes is that there are really two ways to use them. Some class members can be used without creating an object first. These are called shared members, and they are accessed by class name. For example, you can use the shared property DateTime.Now to retrieve a DateTime object that represents the current date and time. You don't need to create a DateTime instance 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 instance. These members have no meaning without a live object and some valid data to draw on.

```
' Get the current date using a shared method.
' Note that you need to use the class name: DateTime.
Dim MyDate As DateTime = 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 shared (C# programmers use the word *static* instead). Shared methods are a major part of the .NET framework, and you will make frequent use of them in this book. Remember, some classes may consist entirely of shared members, and some may use only instance members. Other classes, like DateTime, can provide a combination of the two.

In the next example, which introduces a basic class, we'll use only instance members. This is the default and a good starting point.

A Simple Class

In the next example, we'll build up a simple .NET class bit by bit. First of all, we start by defining the class itself using a special block structure.

VB .NET

```
Public Class MyClass
    ' Class code goes here.
End Class
```

C#

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

You can define as many classes as you need in the same file (which is a departure from Visual Basic 6, which required a separate 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 structure), or they may just be a convenient way to group together related functionality. In this example, our class represents a single product in a company database.

In order to give it some basic functionality, define three private variables that store data about the product: namely, its name, price, and a URL to an image file.

VB .NET

```
Public Class Product
    Private Name As String
    Private Price As Decimal
    Private ImageUrl As String
End Class
```

C#

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

The client can create an object out of this class quite easily. From the client's point of view, the only difference in creating an object variable instead of a class variable is that the New keyword must be specified in order to actually instantiate the class. Otherwise, when you try to use the class you will receive a "null reference" error, because it doesn't actually exist yet.

On the other hand, sometimes you will create an object variable without actually using the New keyword. For example, you might want to assign the variable to an instance that already exists, or you might receive a live object as a return value from a function, and just need a variable to refer to it. In these cases—when you aren't actually creating the class—you shouldn't use the New keyword.

VB .NET

```
Dim SaleProduct As New Product()

' Optionally you could do this in two steps:
' Dim SaleProduct As Product
' SaleProduct = New Product()

' Now release the class from memory.
SaleProduct = Nothing
```

C#

```
Product saleProduct = new Product();

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

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

Class Creation Changes from Visual Basic 6

In Visual Basic 6, it was a bad idea to define an object with the New keyword in the definition. This would not actually create the class, but cause it to be automatically created the first time you refer to it in a code statement. The problem was that you could release the object, try to use the variable again, and end up with a new blank copy of the object automatically recreated. This bizarre behavior has been stamped out in VB .NET. You'll also notice that the Set keyword is no longer needed. VB .NET can distinguish between variables and objects automatically, and does not need your help.

Adding Properties

The simple Product class is essentially useless because there's no way your code can manipulate it. All its information is private and internal. You could make the member variables public, but that could lead to problems because it would allow the client code free access to change any variables, even applying invalid or inconsistent data. Instead, you need to add a "control panel" through which your code can manipulate Product objects. You do this by adding property procedures.

Property procedures have two parts: one procedure that allows the class user to retrieve data, and another part that allows the class user to set data. Property procedures are similar to any other type of procedure in that you can write as much code as you need. For example, you could raise an error to alert the client code of invalid data and prevent the change from being applied. Or you could change multiple internal variables to match the public property. In the Product class example, the property procedures just provide straight access to the internal variables.

To prevent confusion between the internal variables and the property names, the internal variables have been renamed to start with an underscore in the VB .NET code. This is a common convention, but it doesn't represent the only possible approach. In the C# code, the names are differentiated by case.

VB .NET

```
Public Class Product
    Private _Name As String
    Private _Price As Decimal
    Private _ImageUrl As String

    Public Property Name() As String
        Get
            Return _Name
        End Get
        Set(Value As String)
            _Name = Value
        End Set
```

```
    End Property

    Public Property Price() As Decimal
        Get
            Return _Price
        End Get
        Set(Value As Decimal)
            _Price = Value
        End Set
    End Property

    Public Property ImageUrl() As String
        Get
            Return _ImageUrl
        End Get
        Set(Value As String)
            _ImageUrl = Value
        End Set
    End Property

End Class
```

C#

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

The client can now create and configure the class by using its properties and the familiar dot syntax.

VB .NET

```
Dim SaleProduct As New Product()
SaleProduct.Name = "Kitchen Garbage"
SaleProduct.Price = 49.99
SaleProduct.ImageUrl = "http://mysite/garbage.png"
```

C#

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

A Basic Method

You can easily add a method to a class—all you need to do is add a public function or subroutine to the class. The class user can then call this method.

For example, the Product class could have a special GetHtml method that returns a string representing a formatted block of HTML. This HTML can be placed on a web page to represent the product.

VB .NET

```
Public Class Product
    ' (Variables and property procedures omitted for clarity.)

    Public Function GetHtml() As String
        Dim HtmlString As String
        HtmlString = "<h1>" & _Name & "</h1><br>"
        HtmlString &= "<h3>Costs: " & _Price.ToString() & "</h3><br>"
        HtmlString &= "<img src=" & _ImageUrl & ">"
        Return HtmlString
    End Function

End Class
```

C#

```
public class Product
{
    // (Variables and property procedures omitted for clarity.)

    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 internal data, and format it in some attractive way. This really targets our class as a user interface class rather than a pure data class or "business object." Your code can use this method to create a dynamic web page in ASP style, using Response.Write. A web page created in this way is shown in Figure 3-2.

Figure 3-2: Output generated by a Product object

VB .NET

```
Dim SaleProduct As New Product()
SaleProduct.Name = "Kitchen Garbage"
SaleProduct.Price = 49.99
SaleProduct.ImageUrl = "http://mysite/garbage.png"
Response.Write(SaleProduct.GetHtml())
```

C#

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

The interesting thing about the GetHtml method is that it is similar to how an ASP.NET web control works (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 something conceptually similar). It then sends the completed page to the user.

When using a web control, you only see the public interface made up of properties, methods, and events. However, understanding how class code actually works will help you master advanced development. Incidentally, the .NET class library is written using C#, although you can use its classes in any .NET language with equal ease.

A Basic Event

Classes can also use events to notify your code. ASP.NET uses a new event-driven programming model, and you'll soon become very used to writing code that reacts to events. Unless you are creating your own components, however, you won't need to fire your own custom events.

As an illustration, the Product class example has been enhanced with a NameChanged event that occurs whenever the Name is modified through the property procedure. (Note that this event won't fire if code inside the class changes the underlying private name variable without going through the property procedure.)

VB .NET

```
Public Class Product
    ' (Additional class code omitted for clarity.)

    ' Define the event.
    Public Event NameChanged()

    Public Property Name() As String
        Get
            Return _Name
        End Get
        Set(Value As String)
            _Name = Value
            ' Fire the event to all listeners.
            RaiseEvent NameChanged()
        End Set
    End Property

End Class
```

C#

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

    // Define the delegate that represents the event.
    public delegate void NameChangedEventHandler();
    // Define the event.
    public event NameChangedEventHandler NameChanged;

    public string Name
    {
        get
        { return name; }
        set
        {
            name = value;
            // Fire the event to all listeners.
            NameChanged();
        }
    }
}
```

The C# and VB .NET syntax differ slightly. Visual Basic allows you to define the event and any arguments it might have using the special Event keyword, and then fire it with the RaiseEvent statement. C#, on the other hand, requires that you first create a delegate that represents the type of event you are going to use (and its parameters). Then you can define an event. When you fire an event in C#, you just use it by name, you do not need any special function.

VB .NET and C# event handling also differs slightly. Both languages allow you to connect an event to an event handler dynamically, but the syntax differs. Consider the example for our Product class:

VB .NET

```
Dim SaleProduct As New Product()

' This connects the saleProduct.NameChanged event to an event handling
' procedure called ChangeDetected.
AddHandler SaleProduct.NameChanged AddressOf(ChangeDetected)

' Now the event will occur in response to this code:
```

```
SaleProduct.Name = "Kitchen Garbage"
```

C#

```
Product saleProduct = new Product();

// This connects the saleProduct.NameChanged event to an event handling
// procedure called ChangeDetected.
// Note that ChangedDetected needs to match the
// NameChangedEventHandler delegate.
saleProduct.NameChanged += new NameChangedEventHandler(ChangeDetected);

// Now the event will occur in response to this code:
saleProduct.Name = "Kitchen Garbage";
```

The event handler is just a simple procedure called ChangeDetected:

VB .NET

```
Public Sub ChangeDetected()
    ' This code executes in response to the NameChanged event.
End Sub
```

C#

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

If you're using C#, that's essentially the end of the story. If you are a VB .NET programmer, you have an additional option. You don't need to connect events dynamically, instead you can connect them declaratively with the WithEvents keyword and the Handles clause. You'll see this technique in our web form examples when we introduce server control events in next few chapters.

Event Handling Changes from Visual Basic 6

In traditional Visual Basic programming, events were connected to event handlers based on the procedure name. In .NET, this clumsy system is abandoned. Your event handler can have any name you want, and it can even be used to handle more than one event, provided they pass the same type of information in their parameters.

Constructors

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 tried to use a method that relied on some of the data that hasn't been supplied. In the past, this was a major limitation of classes in Visual Basic programming. Today, .NET brings VB up to the level of other modern languages by introducing constructors.

A constructor is just a special method that automatically runs when the class is first created. In C#, this method always has the same name as the name of the class, by convention. In VB .NET, the constructor always has the name New.

If you don't create a constructor, .NET supplies a default constructor that does nothing. If,

however, you create a constructor, the client code must use it. This restriction is useful because you can create constructors that require specific parameters. If the calling code tries to create an object without specifying valid values for all these parameters, you can raise an error and refuse to create the object.

As a quick example, consider the new version of the Product class shown next. It adds a constructor that requires the product price and name as arguments.

VB .NET

```
Public Class Product
   ' (Additional class code omitted for clarity.)

   Public Sub New(name As String, price As Decimal)
      ' These parameters have the same name as the property
      ' procedures, but that doesn't cause a problem.
      ' The local variables have priority.
      _Name = name
      _Price = price
   End Sub

End Class
```

C#

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

   public void Product(string name, decimal price)
   {
      // These parameters have the same name as the
      // internal variables. The "this" keyword is used to refer
      // to the class variables. "this" is the same as "Me" in VB:
      // it refers to the current class.
      this.name = name;
      this.price = price;
   }
}
```

To create a Product object, you need to supply information for these parameters, ensuring it will start its life with valid information.

VB .NET

```
Dim SaleProduct As New Product("Kitchen Garbage", 49.99)
```

C#

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

Most of the classes you use will have special constructors. As with any other method, constructors can be overloaded with multiple versions, each providing a different set of parameters. You can choose the one that suits you best based on the information you have. The .NET framework classes use this technique extensively.

VB .NET Constructors Don't Use the Overloads Keyword

When creating multiple constructors in VB .NET, you don't use the Overloads keyword that was introduced in Chapter 2. There's no technical reason for this discrepancy; it's just a harmless

C#

```
// Create two empty variables (don't use the New keyword).
Product theProduct;
TaxableProduct theTaxableProduct;

// This works, because TaxableProduct derives from Product.
theProduct = new TaxableProduct();

// This will be true.
if (theProduct == TaxableProduct)
{
    // Convert the object and assign to the other variable.
    theTaxableProduct = (TaxableProduct)theProduct;
}

decimal totalPrice;

// This works.
totalPrice = theTaxableProduct.TotalPrice;

// This won't work, even though theTaxableProduct and theProduct
// are the same object.
// The Product class does not provide a TotalPrice property.
totalPrice = theProduct.TotalPrice;
```

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. These topics aren't covered in this book, and they don't affect most ASP.NET programmers.

Understanding Namespaces and Assemblies

Whether you realize it at first or not, every piece of code in .NET exists inside a class. In turn, every class exists inside a namespace. Figure 3-3 shows this arrangement for your own code and the DataTime object. Keep in mind that this is an extreme simplification—the System namespace alone is stocked with several hundred classes.

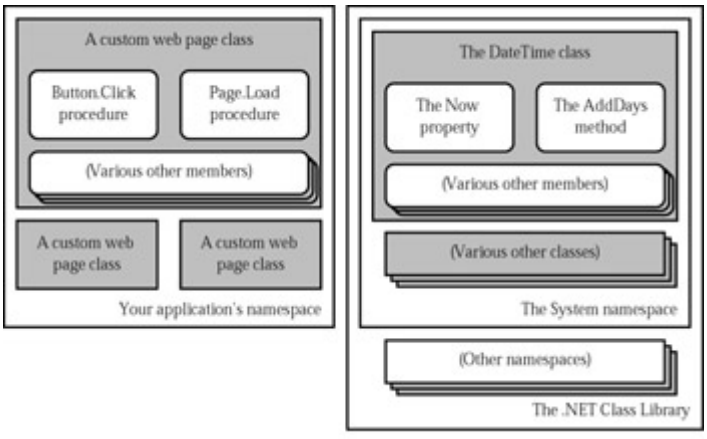


Figure 3-3: .NET namespaces

Namespaces represent the single organizing principle used to group all the different types in the class library. Without namespaces, these types would all be grouped into a single long and messy list, much like the disorganized Windows API. This type of organization is practical for a small set of information, but it would be useless for the thousands of types included with .NET.

Many of the chapters in this book introduce you to one of .NET's 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 MSDN reference, which painstakingly documents the properties, methods, and events of every class in every namespace (see Figure 3-4).

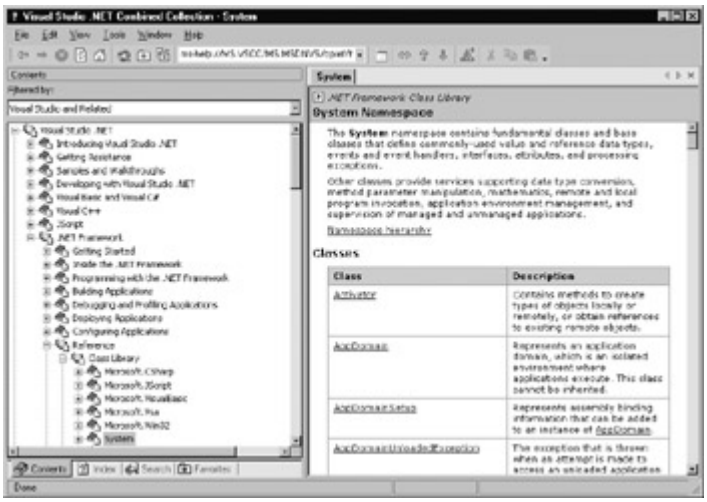


Figure 3-4: The MSDN Class Library reference

Often when you write ASP.NET code, you will just use the default namespace. If, however, you want to create a more specific namespace, you can do so using a simple block structure:

VB .NET

```
Namespace MyCompany
   Namespace MyApp
      Public Class Product
         ' Code goes here.
      End Class
   End Namespace
End Namespace
```

C#

```
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 technology developers without worrying about a name collision. If they 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 will then almost certainly be unique.

Namespaces don't take a private or public access keyword, and can be stacked as many layers deep as you need.

Importing Namespaces

Having to type long fully qualified names is certain to tire out the fingers and create overly verbose code. To tighten things up, it's standard practice to import the namespaces you want to use. When you import a namespace, you don't need to type the fully qualified name. Instead, you can use the object as though it is defined locally.

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

VB .NET

```
Imports MyCompany.MyApp
```

C#

```
using MyCompany.MyApp;
```

Consider the situation without importing a namespace:

VB .NET

```
Dim salesProduct As New MyCompany.MyApp.Product()
```

C#

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

It's much more manageable when you import the MyCompany.MyApp namespace:

VB .NET

```
Dim salesProduct As New Product()
```

C#

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

Assemblies

You might wonder what gives you the ability to use the class library namespaces in a .NET program. Are they hard-wired directly into the language? The truth is that all .NET classes are contained in assemblies, which is the .NET equivalent of traditional .exe and .dll files. There are two ways you can use an assembly:

- By placing it in the same directory or in a subdirectory of your application. The CLR will automatically examine all these assemblies and make their classes available to your application. No additional steps or registration are required.

- By placing it in the Global Assembly Cache (GAC), which is a systemwide store of shared assemblies. You won't take this option for your own components unless you are a component vendor. All the .NET classes, however, are a part of the GAC, because it wouldn't make sense to install them separately in every application directory. You'll learn more about assemblies in Chapter 21.

Assemblies and the .NET Classes

The .NET classes are actually contained in a number of different assemblies. For example, the basic types in the System namespace come from the mscorlib.dll assembly. Many ASP.NET types are found in the System.Web.dll assembly. If you are precompiling your ASP.NET pages, you will need to explicitly reference these assemblies. This topic is covered in Chapter 5.

```
    <div>

      Convert:  
      <input type="text">  US dollars to Euros.<br><br>
      <input type="submit" value="OK">

    </div>
  </form>
</body></HTML>
```

The Problem with Response.Write

In a traditional ASP program, you would add the currency conversion functionality to this page by examining the posted form values and manually writing the result to the end of the page with the Response.Write command. This approach works well for a simple page, but it encounters a number of difficulties as the program becomes more sophisticated:

"Spaghetti" code The order of the Response.Write statements determines the output. If you want to tailor different parts of the output based on a condition, you will need to reevaluate that condition at several different places in your code.

Lack of flexibility Once you have perfected your output, it's very difficult to change it. If you decide to modify the page several months later, you have to read through the code, follow the logic, and try to sort out numerous details.

Confusing content and formatting Depending on the complexity of your user interface, your Response.Write may need to add new HTML tags and style attributes. This encourages programs to tangle formatting and content details together, making it difficult to change just one or the other at a later date.

Complexity Your code becomes increasingly intricate and disorganized as you add different types of functionality. For example, it could be extremely difficult to track the effects of different conditions and different Response.Write blocks if you created a combined tax-mortgage-interest calculator.

Quite simply, an ASP.NET application that needs to create a sizable portion of interface using Response.Write commands encounters the same dilemmas that a Windows program would find if it needed to manually draw its text boxes and command buttons on an application window in response to every user action.

Server Controls

In ASP.NET, you can still use Response.Write to create a dynamic web page. But ASP.NET provides a better approach. It allows you to turn static HTML tags into objects (called controls) that you can program on the server.

ASP.NET provides two sets of server controls:

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

- **Web controls** are similar to the HTML server controls, but provide a richer object model with a variety of properties for style and formatting details, more events, and a closer parallel to Windows development. Web controls also feature some user interface elements that have no HTML equivalent, such as the DataGrid and validation controls. We examine web

controls in the next chapter.

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 updated automatically when the page is rendered and sent to the client.

They retain their state. You can write your program the same way you would write a traditional Windows program. There's no need to recreate a web page from scratch each time you send it to the user.

They fire events. Your code can respond to these events, just like ordinary controls in a Windows application. In ASP code, everything is grouped into one block that executes from start to finish. With event-based programming, you can easily respond to individual user actions and create more structured code.

To convert the currency converter to an ASP.NET page that uses server controls, all you need to do is add the special 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 will use to refer to the control in code.

In the currency converter application, the input text box and submit button can be changed into server controls. In addition, the <form> element must also be processed as a server control to allow ASP.NET to access the controls it contains.

```
<HTML><body>
  <form method="post" runat="server">
    <div>

      Convert:  
      <input type="text" id="US" runat="server">  US dollars to
      Euros.
      <br><br>
      <input type="submit" value="OK" id="Convert" runat="server">

    </div>
  </form>
</body></HTML>
```

Viewstate

If you look at the HTML that is sent to you when you request the page, you'll see it is slightly different than the information in the .aspx file. First of all, the runat="server" attributes are stripped out (as they have no meaning to the client browser, which can't interpret them). More importantly, an additional hidden field has been added to the form.

```
<HTML><body>
  <form method="post" runat="server">
  <input type="hidden" name="__VIEWSTATE"
  value="dDw3NDg2NTI5MDg7Oz4=" />
    <div>

      Convert:  
      <input type="text" id="US" runat="server">  US dollars to
      Euros.
      <br><br>
      <input type="submit" value="OK" id="Convert" runat="server">
```

```
   </div>
  </form>
</body></HTML>
```

This hidden field stores information about the state of every control in the page in a compressed and lightly encrypted form. It allows you to manipulate control properties in code and have the changes automatically persisted. This is a key part of the web forms programming model, which allows you to forget about the stateless nature of the Internet, and treat your page like a continuously running application.

ASP.NET Controls Maintain State Automatically

Even though the currency converter program does not yet include any code, you will 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 will be cleared every time the page is posted back.)

The HTML Control Classes

Before you can continue any further with the currency calculator, you need to know about the control objects you have created. All the HTML server controls are defined in the System.Web.UI.HtmlControls namespace. There is a separate class for each kind of control. Table 6-1 describes the basic HTML server controls.

So far, the currency converter defines three controls, which are instances of the HtmlForm, HtmlInputText, and HtmlInputButton classes, respectively. It's important that you know the class names, because you need to define each control class in the code-behind file if you want to interact with it. (Visual Studio .NET simplifies this task: whenever you add a control using its web designer, the appropriate tag is added to the .aspx file and the appropriate variables are defined in the code-behind class.)

Table 6-1: The HTML Server Control Classes

Class Name	HTML Tag Represented
HtmlAnchor	<a>
HtmlButton	<button>
HtmlForm	<form>
HtmlImage	<img>
HtmlInputButton	<input type="button">, <input type="submit">, and <input type="reset">
HtmlInputCheckBox	<input type="checkbox">
HtmlInputControl	<input type="text">, <input type="submit">, and <input type="file">
HtmlInputFile	<input type="file">
HtmlInputHidden	<input type="hidden">
HtmlInputImage	<input type="image">
HtmlInputRadioButton	<input type="radio">
HtmlInputText	<input type="text"> and <input type="password">

HtmlSelect	<select>
HtmlTable, HtmlTableRow, and HtmlTableCell	<table>, <tr>, <th> and <td>
HtmlTextArea	<textarea>
HtmlGenericControl	Any other HTML element

The HTML Server Control Reference

This chapter introduces many new server controls. As we work our way through the examples, I'll explain many of the corresponding properties and events, and the class hierarchy. However, for single-stop shopping, you are encouraged to refer to Chapter 26, which provides a complete HTML server control reference. Each control is featured separately, with a picture, example tag, and a list of properties, events, and methods. In the meantime, Table 6-2 gives a quick overview of some important properties.

Table 6-2: Important Properties

Control	**Most Important Properties**
HtmlAnchor	Href, Target, Title
HtmlImage and HtmlInputImage	Src, Alt, Width, and Height
HtmlInputCheckBox and	CheckedHtmlInputRadioButton
HtmlInputText	Value
HtmlSelect	Items (collection)
HtmlTextArea	Value
HtmlGenericControl	InnerText

Events

To actually add some functionality to the currency converter, you need to add some ASP.NET code. Web forms are event-driven, which means that every piece of code acts in response to a specific event. In the simple currency converter page case, the event is when the user clicks the submit button (named Convert). The HtmlInputButton class provides this as the ServerClick event.

Before continuing, it makes sense to add another control that can be used to display the result of the calculation (named Result). The example now has four server controls:

- A form (HtmlForm object). This is the only control you do not need to use in your code-behind class.
- An input text box named US (HtmlInputText object).
- A submit button named Convert (HtmlInputButton object).
- A div tag named Result (HtmlGenericControl object).

CurrencyConverter.aspx

```
<%@ Page Language="VB" Inherits="CurrencyConverter"
    Src="CurrencyConverter.vb"
    AutoEventWireup="False" %>
```

Chapter 7: Web Controls

Overview

The last chapter introduced one of ASP.NET's most surprising revolutions: the change to event-driven and control-based programming. This change allows you to create programs for the Internet following the same structured, modern style you would use for a Windows application.

However, HTML server controls really only show a glimpse of what is possible with ASP.NET's new control model. To see some of the real advantages provided by this shift, you need to explore web controls, which provide a rich, extensible set of control objects. In this chapter, we'll examine the basic web controls and their class hierarchy. We'll also delve deeper into ASP.NET's event handling, and examine the web page lifecycle.

Stepping Up to Web Controls

Now that you've seen the new model of server controls, you might wonder why we need additional web controls. But in fact, HTML controls are still more limited than server controls need to be. For example, every HTML control corresponds directly to an HTML tag, meaning that you are bound by the limitations and abilities of the HTML language. Web controls, on the other hand, emphasize the future of web design:

They provide rich user interface. A web control is programmed as an object, but doesn't necessarily correspond to a single tag in the final HTML page. For example, you might create a single Calendar or DataGrid 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 anything about HTML. The control creates the required HTML tags for you.

They provide a consistent object model. The HTML language 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. Controls that display text, whether it is a caption or a user-editable text box, 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 DHTML. You don't need to know about the client because ASP.NET handles that layer and automatically uses the best possible set of features.

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.

The Examples in this Book Use Web Controls

Throughout this book, I'll show examples that use the full set of web controls. In order to master ASP.NET development, you need to become comfortable with these user interface ingredients and understand all their abilities. HTML server controls, on the other hand, are less important for web development, unless you need to have a fine-grained control over the HTML code that will be

generated and sent to the client. They are de-emphasized.

Basic Web Control Classes

If you have created Windows applications before, you are probably familiar with the basic set of standard controls, including labels, buttons, and text boxes. ASP.NET provides web controls for all these standbys. (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 framework to another.)

Table 7-1 lists the basic control classes, and the HTML elements they generate. Note that some controls, such as Button and TextBox, can be rendered as different HTML elements. ASP.NET uses the element that matches the properties you have set. Also, some controls have no single HTML equivalent. For example, the CheckBoxList and RadioButtonList controls output as a <table> that contains multiple HTML checkboxes or radio buttons. ASP.NET wraps them into a single object on the server side for convenient programming, illustrating one of the primary strengths of web controls.

Table 7-1: 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 <img> tag
ImageButton	<input type="image">
Image	<img>
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
Panel	
Table, TableRow, and TableCell	<table>, <tr>, and <td> or <th>

This table omits some of the more specialized rich controls, which we'll see in Chapter 9, and the advanced data controls, which we'll see in Chapter 15.

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 />. Any attributes in the tag correspond to a control property, aside from the runat="server" attribute, which declares that the control will 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.

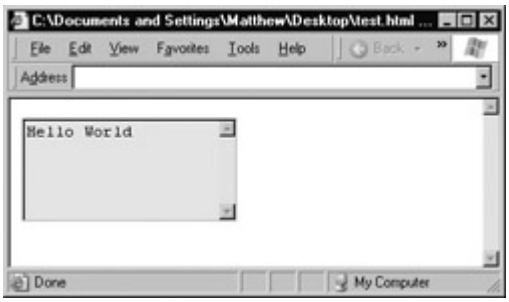
```
<input type="text" name="ctrl0" />
```

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, TextBox.TextMode property allows you to specify SingleLine (the default), MultiLine (for a text area type of control), or Password (for an input control that displays all asterisks when the user types in a value).

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

The resulting HTML uses the text area element and sets all the required style attributes.

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



Clearly, it's easy to create a web control tag. However, you need to understand the control class and the properties that are available to you.

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, as shown in Figure 7-1.

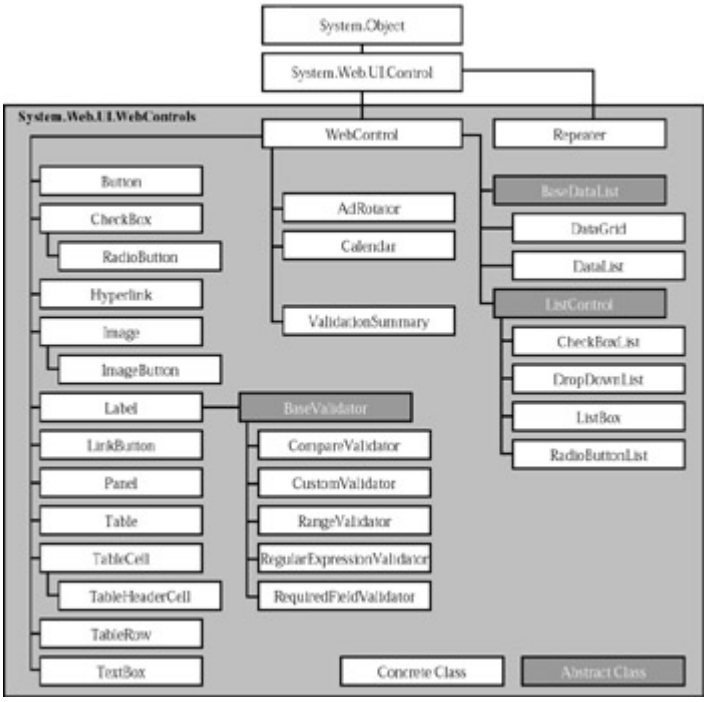


Figure 7-1: Web control hierarchy

This inheritance diagram includes some controls that we won't examine in this chapter, including the data controls such as the Repeater, DataList, and DataGrid, and the validation controls.

The WebControl Base Class

All web controls begin by inheriting from the WebControl base class. This class defines essential functionality for tasks such as data binding, and includes some basic properties that you can use with any control (see Table 7-2).

Table 7-2: 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. This feature is only supported on Internet Explorer 4.0 and higher.
BackColor, BorderColor and ForeColor	Sets the colors used for the background, foreground, and border of the control. Usually, the foreground color is used for text.
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 may need to cast the reference with the CType function to access control-specific properties.
Enabled	When set to False, the control will be visible, but 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 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 special System.Drawing.Font 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.
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 lowest TabIndex, provided it is enabled. This property is only supported in Internet Explorer 4.0 and higher.
ToolTip	Displays a text message when the user hovers the mouse above the control. Many older browsers do not 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.

Units

All the properties that use measurements, including BorderWidth, Height, and Width, use the special Unit structure. When setting a unit in a control tag, make sure you add append px (pixel) or % (for percentage) to the number to indicate the type of unit.

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

If you are assigning one of these values at runtime, you need to use one of the shared properties of the Unit type.

```
' 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 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
Dim MyUnit as New Unit(300, UnitType.Pixel)

' Assign the Unit object to several controls or properties.
pnl.Height = MyUnit
pnl.Width = MyUnit
```

Enumerated Values

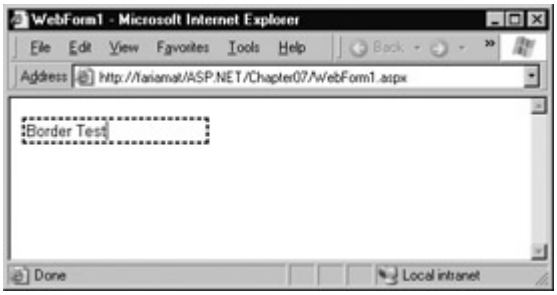
Enumerations are used heavily in the .NET class library to group together a set of related

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 using the dot syntax.

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

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

```
<asp:TextBox BorderStyle="Dashed" Text="Border Test" id="txt"
runat="server" />
```



Colors

The Color property refers to a Color object from the System.Drawing namespace. Color objects can be created in several different ways:

> **Using an ARGB (alpha, red, green, blue) color value.** You specify each value as an integer.
>
> **Using a predefined .NET color name.** You choose the correspondingly named read-only property from the Color class. These properties include all the HTML colors.
>
> **Using an HTML color name.** You specify this value as a string using the ColorTranslator class.

To use these techniques, you must import the System.Drawing namespace.

```
Import System.Drawing
```

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

```
' Create a color from an ARGB value
Dim Alpha As Integer = 255, Red As Integer = 0
Dim Green As Integer = 255, Blue As Integer = 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="Border Test" id="txt"
runat="server" />
```

The color names that you can use are listed in Chapter 27. Alternatively, you can use a

hexadecimal color number (in the format #<red><green><blue>) as shown next.

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

Fonts

The Font property actually references a full FontInfo object (which is defined in the System.Drawing namespace). Every FontInfo object has several properties that define its name, size, and style (see Table 7-3).

Table 7-3: FontInfo Properties

Property	Description
Name	A string indicating the font name (such as "Verdana").
Size	The size of the font as a FontUnit object. This can represent an absolute or relative size.
Bold, Italic, Strikeout, Underline, and Overline	Boolean properties that either apply the given style attribute or ignore it.

In code, you can assign to the various font properties using the familiar dot syntax.

```
ctrl.Font.Name = "Verdana"
ctrl.Font.Bold = "True"
```

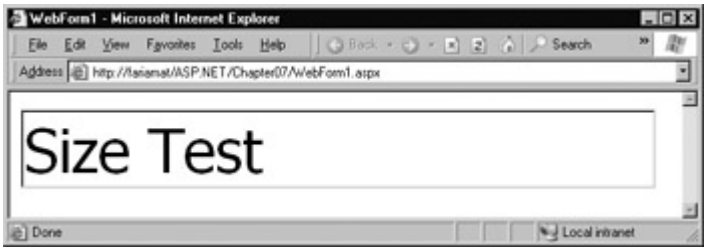
You can set the size using the FontUnit type.

```
' 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 with a relative size:

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

List Controls

The simple list controls (ListBox, DropDownList, CheckBoxList, and RadioButtonList) are extremely easy to use. Like the HtmlSelect control described in the previous chapter, they provide

a SelectedIndex property that indicates the selected row as a zero-based index. (For example, if the first item in the list is selected, ctrl.SelectedIndex will be 0.) 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 in the HTML code), and Selected (True or False depending on whether the item is selected).

In the last chapter, we used code like this to retrieve the selected ListItem object from an HtmlSelect control called Currency.

```
Dim Item As ListItem
Item = Currency.Items(Currency.SelectedIndex)
```

With a web control, this can be simplified with a clearer syntax.

```
Dim Item As ListItem
Item = Currency.SelectedItem
```

Multiple-Select List Controls

Some list controls can allow multiple selections. This isn't possible for DropDownList or RadioButtonList, but it is possible for a ListBox if you have set the SelectionMode property to the enumerated value ListSelectionMode.Multiple. With the CheckBoxList, multiple selections are always possible. To find all the selected items, you need to iterate through the Items collection of the list control, and check the ListItem.Selected property of each item.

Figure 7-2 shows a simple web page example. It provides a list of computer languages and indicates which selections the user made when the OK button is clicked.

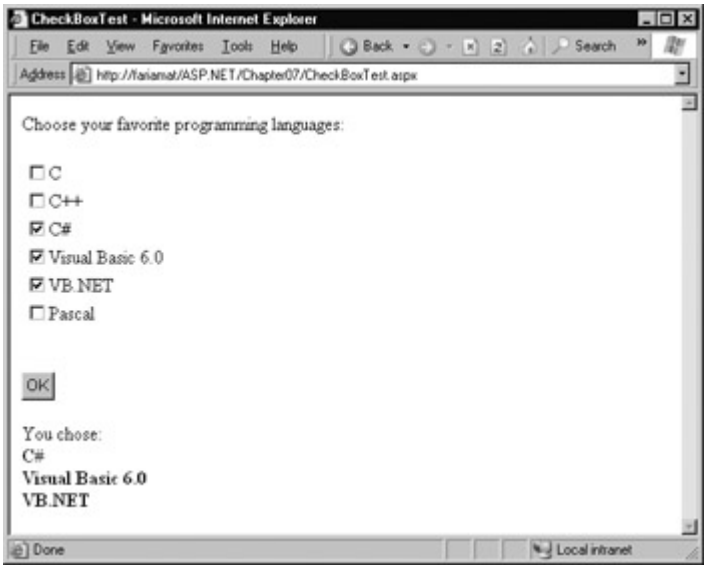


Figure 7-2: A simple CheckListBox test

The .aspx file defines CheckListBox, Button, and Label controls.

```
<%@ Page Language="VB" Inherits="CheckListTest" Src="CheckListTest.vb"
    AutoEventWireup="False" %>

<HTML><body>
  <form method="post" runat="server">

  Choose your favorite programming languages:<br><br>
  <asp:CheckBoxList id="chklst" runat="server" /><br><br>
  <asp:Button id="cmdOK" Text="OK" runat="server" /><br><br>
  <asp:Label id="lblResult" runat="server" />
```

```
  </form>
</body></HTML>
```

The code adds items to the CheckListBox at startup, and iterates through the collection when the button is clicked.

```
Imports System
Imports System.Web
Imports System.Web.UI
Imports system.Web.UI.WebControls
Imports System.Web.UI.HtmlControls

Public Class CheckListTest
  Inherits Page

    Protected chklst as CheckBoxList
    Protected WithEvents cmdOK As Button
    Protected lblResult as Label

    Private Sub Page_Load(sender As Object, e As EventArgs) _
      Handles MyBase.Load
        If Me.IsPostBack = False
            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")
        End If
    End Sub

    Private Sub cmdOK_Click(sender As Object, e As EventArgs) _
      Handles cmdOK.Click

        lblResult.Text = "You chose:<b>"

        Dim lstItem As ListItem
        For Each lstItem In chklst.Items
            If lstItem.Selected = True Then
                ' Add text to label.
                lblResult.Text &= "<br>" & lstItem.Text
            End If
        Next

        lblResult.Text &= "</b>"

    End Sub

End Class
```

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) follow this convention to make user interface code as clear as possible. Some recommended control prefixes are as follows:

- Button: cmd
- CheckBox: chk

- Image: img
- Label: lbl
- List control: lst
- Panel: pnl
- RadioButton: opt
- TextBox: txt

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

Table Controls

Essentially, the Table object contains a hierarchy of objects. Each Table object contains one or more TableRow objects. Each TableRow object contains one or more TableCell object, and each TableCell object contains other ASP.NET controls that display information. If you're familiar with the HTML table tags, this relationship (shown in Figure 7-3) will seem fairly logical.

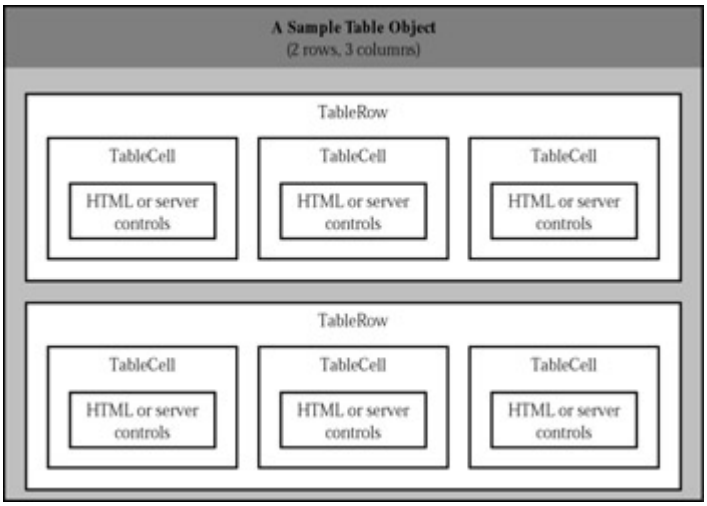


Figure 7-3: Table control containment

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 7-4. It allows the user to specify a number of rows and columns, and an option that specifies whether cells should have borders.

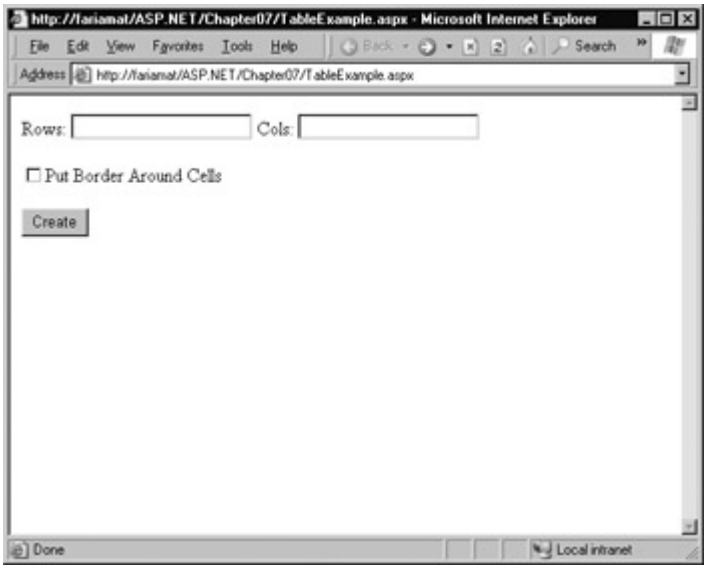


Figure 7-4: The table test options

When the user clicks the generate button, the table is filled dynamically with sample data according to the selected options, as shown in Figure 7-5.

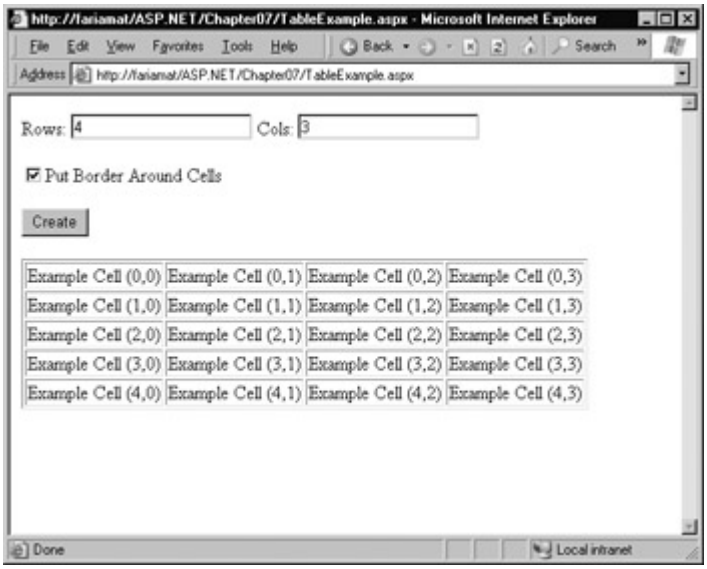


Figure 7-5: A dynamically generated table

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

```
<%@ Page Language="VB" Inherits="TableTest" Src="TableTest.vb"
    AutoEventWireup="False" %>

<HTML><body>
  <form method="post" runat="server">

    Rows:
    <asp:TextBox id="txtRows" runat="server" /> 
    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" runat="server" Text="Create" /><br><br>
    <asp:Table id="tbl" runat="server" />

  </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 Text="A Test Row" id="cell" runat="server">
      <!-- Instead of using the Text property, you could add other
           ASP.NET control tags here. -->
    </asp:TableCell>
  </asp:TableRow>
</asp:Table>
```

In the table test web page, there are no nested elements. That means the table will be created as a server-side control object, but unless the code adds rows and cells it will not be rendered in the final HTML page.

The TablePage 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 Class TableTest
  Inherits Page

    Protected txtRows As TextBox
    Protected txtCols As TextBox
    Protected tbl As Table
    Protected chkBorder As CheckBox
    Protected WithEvents cmdCreate As Button

    Private Sub Page_Load(sender As Object, e As EventArgs) _
      Handles MyBase.Load
        ' 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)
    End Sub

    Private Sub cmdCreate_Click(sender As Object, e As EventArgs) _
      Handles cmdCreate.Click

        ' Remove all the current rows and cells.
        ' This would not be necessary if you set
        ' EnableViewState = False.
        tbl.Controls.Clear()

        Dim i, j As Integer
        For i = 0 To Val(txtRows.Text)

            ' Create a new TableRow object.
            Dim rowNew As New TableRow()

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

            For j = 0 To Val(txtCols.Text)

                ' Create a new TableCell object.
                Dim cellNew As New TableCell()
                cellNew.Text = "Example Cell (" & i.ToString() & ","
                cellNew.Text &= j.ToString & ")"

                If chkBorder.Checked = True Then
                    cellNew.BorderStyle = BorderStyle.Inset
```

```
            cellNew.BorderWidth = Unit.Pixel(1)
        End If
        ' Put the TableCell in the TableRow.
        rowNew.Controls.Add(cellNew)

    Next
  Next

 End Sub

End Class
```

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 special 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 example uses this technique (see Figure 7-6).

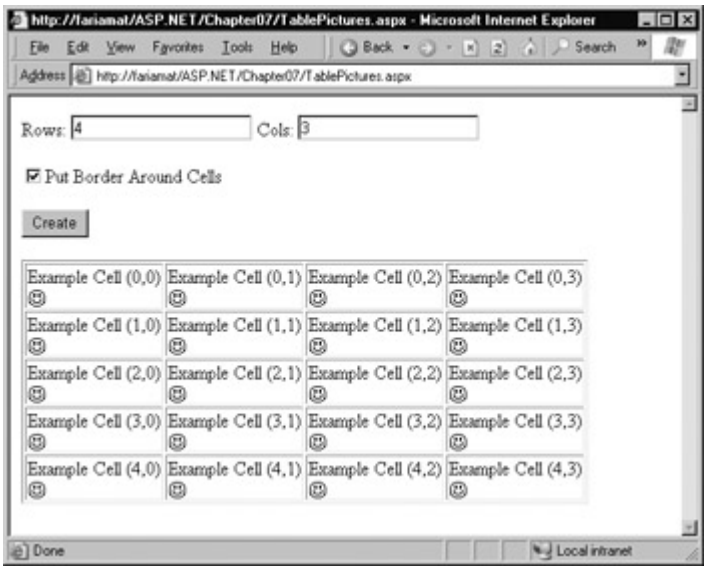


Figure 7-6: A table with contained controls

```
' Create a new TableCell object.
Dim cellNew As New TableCell()

' Create a new Label object.
Dim lblNew As New Label()
lblNew.Text = "Example Cell (" & i.ToString()
lblNew.Text &= "," & j.ToString() & ")<br>"

Dim imgNew As 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 the fact the each TableRow and TableCell are full-featured objects. If you wanted, you could give each cell a different border style, border color, and text color. The full list of TableRow and TableCell properties is described in Chapter 27.

AutoPostBack and Web Control Events

The previous chapter explained that one of the main limitations of HTML server controls is their

limited set of useful events—exactly two. HTML controls that trigger a postback, such as buttons, raise a ServerClick event. Input controls provide a ServerChange event that won't be detected until the page is posted back.

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 (see Figure 7-7).

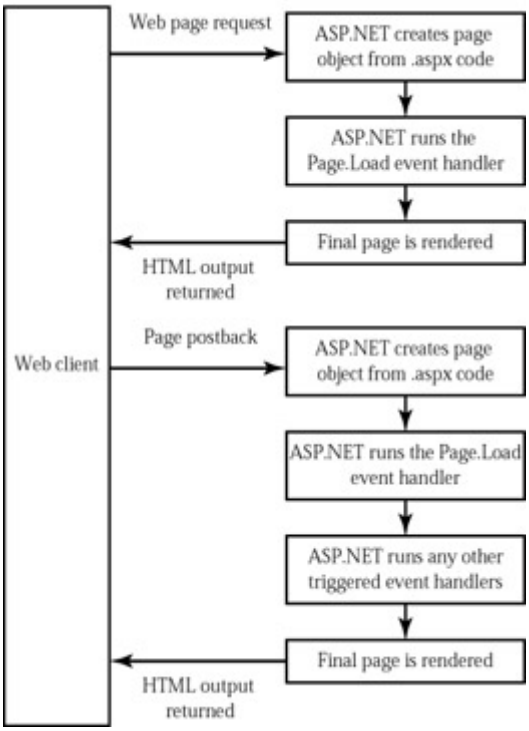


Figure 7-7: The page processing sequence

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 to an event that occurs on the client? The answer is a new innovation called the automatic postback.

Automatic postback submits a page back to the server when it detects a specific user action. This gives your code the chance to run again and create a new, updated page. Controls that support automatic postbacks include almost all input controls. A basic list of web controls and their events is provided in Table 7-4.

Table 7-4: Web Control Events

Event	Web Controls that Provide It
Click	Button, ImageButton
TextChange	TextBox (only fires after the user changes the focus to another control)
CheckChanged	CheckBox, RadioButton
SelectedIndexChanged	DropDownList, ListBox, CheckBoxList, RadioButtonList

If you want to capture a change event for a web control, you need to set its AutoPostBack property to True. That means that when the user clicks a radio button or checkbox, the page will be resubmitted to the server. The server examines the page, loads all the current information, and then allows your code to perform some extra processing before returning the page back to the user.

In other words, every time you need to update the web page, it is actually being sent to the server

and recreated (see Figure 7-8). However, ASP.NET makes this process so transparent that your code can treat your web page like a continuously running program that fires events.

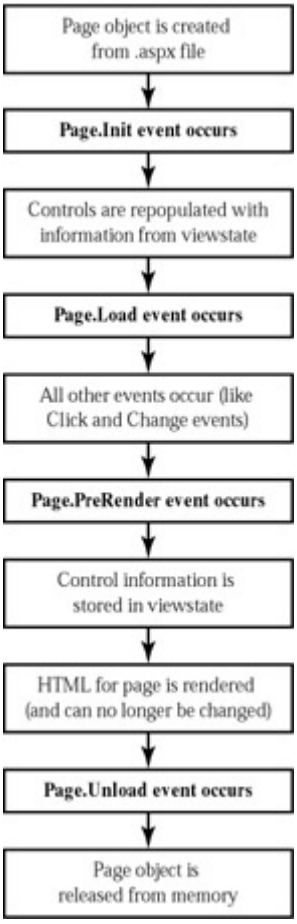


Figure 7-8: The postback processing sequence

This postback system is not 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, are not 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.

How Postback Events Work

The AutoPostBack feature uses a couple of interesting tricks and a well-placed JavaScript function named __doPostBack.

```
<script language="javascript">
<!--
    function __doPostBack(eventTarget, eventArgument) {
        var theform = document.Form1;
        theform.__EVENTTARGET.value = eventTarget;
        theform.__EVENTARGUMENT.value = eventArgument;
        theform.submit();
    }
// -->
</script>
```

ASP.NET adds the JavaScript code to the page automatically if any of your controls are set to use AutoPostBack. It also adds two additional hidden input fields that are used to pass information back to the server: namely, the ID of the control that raised the event, and any additional information that might be relevant.

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

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

Finally, the control that has its AutoPostBack property set to True is connected to the __doPostBack function using the onclick or onchange attribute. The following example shows a list control called lstBackColor, which posts back automatically by calling the __doPostBack function.

```
<select id="lstBackColor" onchange="__doPostBack('lstBackColor','')"
 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. It's possible that you may have created a solution like this manually for traditional ASP programs. The ASP.NET framework handles these details for you automatically, simplifying life a great deal.

The Page Lifecycle

To understand how web control events work, you need to have a solid understanding of the page lifecycle. 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 event is invoked, and the page is resubmitted to the server.

2. ASP.NET recreates the page object using the .aspx file.

3. ASP.NET retrieves state information from the hidden viewstate field, and updates the controls accordingly.

4. The Page.Load event is fired.

5. The appropriate change event is fired for the control.

6. The Page.Unload event fires and the page is rendered (transformed from a set of objects to an HTML page).

7. The new page is sent to the client.

To watch these events in action, it helps to create a simple event tracker application (see Figure 7-9). 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 of events.

Figure 7-9: The event tracker

EventTracker.aspx

```
<%@ Page Language="VB" Src="EventTracker.vb" Inherits="EventTracker"
    AutoEventWireup="False"%>

<HTML><body>
  <form method="post" runat="server">
    <h3>Controls being monitored for change events:</h3>
    <asp:TextBox id=txt runat="server" AutoPostBack="True" /><br><br>
    <asp:CheckBox id=chk runat="server" AutoPostBack="True" /><br><br>
    <asp:RadioButton id=opt1 runat="server" GroupName="Sample"
        AutoPostBack="True" />
    <asp:RadioButton id=opt2 runat="server" GroupName="Sample"
        AutoPostBack="True" /><br><br><br>
    <h3>List of events:</h3>
    <asp:ListBox id=lstEvents runat="server" Width="355px"
        Height="505px" /><br>
  </form>
</body></HTML>
```

EvenTracker.vb

```
Public Class EventTracker
  Inherits System.Web.UI.Page

    Protected WithEvents lstEvents As ListBox
    Protected WithEvents chk As CheckBox
    Protected WithEvents txt As TextBox
    Protected WithEvents opt1 As RadioButton
    Protected WithEvents opt2 As RadioButton

    Private Sub Page_Load(sender As Object, e As EventArgs) _
      Handles MyBase.Load
        Log("<< Page_Load >>")
    End Sub

    Private Sub Page_PreRender(sender As Object, e As EventArgs) _
      Handles MyBase.PreRender
        ' When the Page.UnLoad event occurs it is too late
        ' to change the list.
        Log("Page_PreRender")
    End Sub

    Private Sub CtrlChanged(sender As Object, e As System.EventArgs) _
      Handles chk.CheckedChanged, opt1.CheckedChanged, _
      opt2.CheckedChanged, txt.TextChanged
```

```
        ' Find the control ID of the sender.
        ' This requires converting the Object type into a Control.
        Dim ctrlName As String = CType(sender, Control).ID
        Log(ctrlName & " Changed")
    End Sub

    Private Sub Log(entry As String)
        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
    End Sub

End Class
```

Interesting Facts About this Code

- The code encapsulates the ListBox logging in a Log subroutine, which automatically scrolls to the bottom of the list each time a new entry is added.
- All the change events are handled by the same subroutine, CtrlChanged. The event handling code uses the source parameter to find out what control sent the event and incorporate that in the log string.
- All the controls that use automatic postback have to be defined WithEvents in the Page class, or the Handles clause will not work.

A Simple Web Page Applet

Now that you've had a whirlwind tour of the basic web control model, it's time to put it to work with our second single-page utility. In this case, it's a simple example for a dynamic e-card generator (see Figure 7-10). You could extend this sample (for example, allowing users to store e-cards to a database, or using the techniques in Chapter 16 to mail notification to card recipients), but on its own it represents a good example of basic control manipulation.

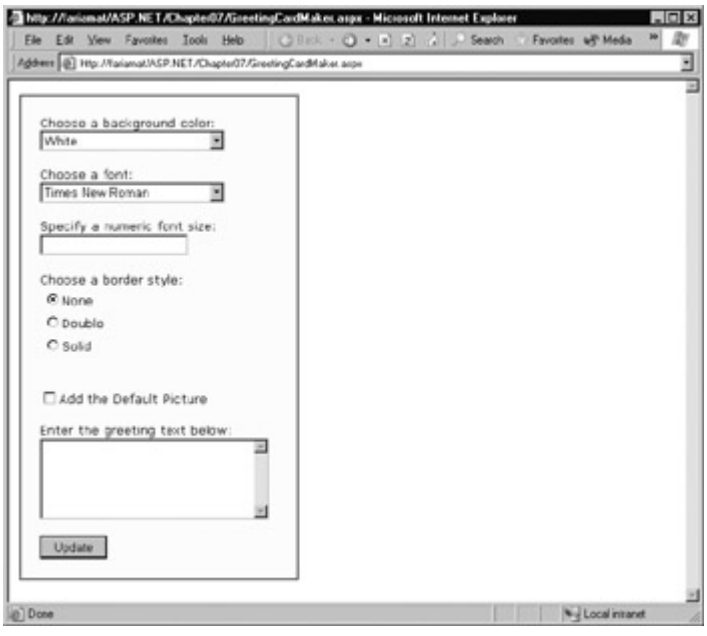


Figure 7-10: The card generator

The page is divided into two regions. On the left is an ordinary tag containing a set of web controls for specifying card options. On the right is a Panel control (named pnlCard), which

UNIT - 3
USING VISUAL .NET
VALIDATION AND RICH CONTROL

Chapter 8: Using Visual Studio .NET

Overview

In the past, ASP developers have overwhelmingly favored simple text editors like Notepad for programming web pages. Other choices were possible, but each one suffered from its own quirks and limitations. Tools like Visual Interdev and Visual Basic web classes were useful for rapid development, but often made deployment more difficult or obscured some important features. In short, the standard was a plain, gloves-off approach of raw HTML text with blocks of code inserted wherever necessary.

Visual Studio .NET provides the first design tool that can change that. First of all, it's extensible, and can even work in tandem with other straight HTML editors like FrontPage or Dreamweaver. It also inherits the best features from editors like Visual Interdev, Visual Basic 6.0, and the Visual Studio IDE for C++ 6.0, including a matchless set of productivity enhancements.

This chapter provides a lightning tour that shows how to create a web application in the Visual Studio .NET environment. You'll also learn how IntelliSense can dramatically reduce the number of errors you'll make, and how to use the renowned single-step debugger that lets you peer under the hood and "watch" your program in action.

The Promise of Visual Studio .NET

All .NET programs are created with text source files. For example, C# programs are stored in .cs files and VB programs in .vb files, regardless of whether they are targeted for the Windows platform, ASP.NET, or the simple command prompt window. Despite this fact, you'll rarely find VB or C# developers creating Windows applications by hand in a text editor. Not only is the process tiring, it also opens the door to a host of possible errors that could be easily caught at design time.

Visual Studio .NET is an indispensable tool for the developers of any platform. It provides impressive benefits:

Integrated error checking Visual Studio .NET can detect a wide range of problems, such as data type conversion errors, missing namespaces or classes, and undefined variables. Errors are detected as you type, then underlined and added to an error list for quick reference.

Web form designer To add ASP.NET controls, you simply drag them to the appropriate location, resize them by hand, and configure their properties. Visual Studio .NET does the heavy lifting: it automatically creates the underlying .aspx template file for you with the appropriate tags and attributes, and adds the control variables to your code-behind file.

Productivity enhancements Visual Studio .NET makes coding quick and efficient, with a collapsible code display, automatic statement completion, and color-coded syntax. You can even create sophisticated macro programs that automate repetitive tasks.

Easy deployment When you start an ASP.NET project in Visual Studio .NET, all the files you need are generated automatically, including a sample web.config file. When you compile the application, all your page classes are compiled into one DLL for easy deployment.

Complete extensibility You can add your own custom add-ins, controls, and dynamic

help plug-ins to Visual Studio .NET, and customize almost every aspect of its appearance and behavior.

Fine-grained debugging Visual Studio .NET's integrated debugger allows you to witness code execution, pause your program at any point, and trace variable contents. These debugging tools can save endless headaches when writing complex code routines.

Starting a Visual Studio .NET Project

You start Visual Studio .NET from the Start menu by selecting Programs | Microsoft Visual Studio .NET 7.0 | Visual Studio .NET 7.0. The IDE will load up with a special start page (see Figure 8-1).

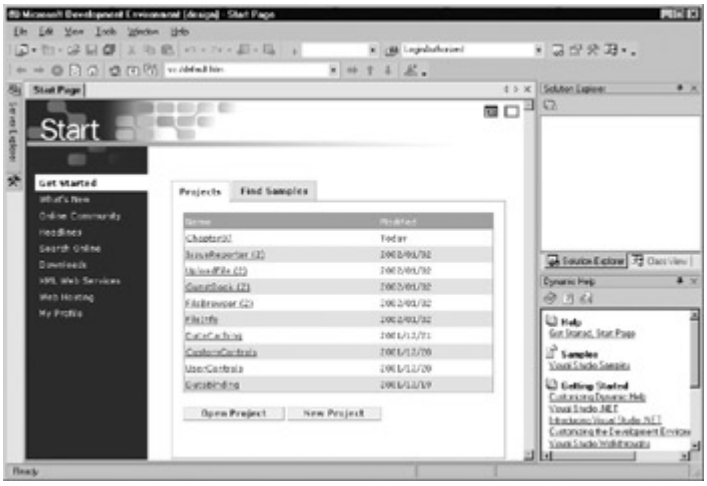


Figure 8-1: The Visual Studio .NET start page

The start page provides some interesting options. Its primary purpose is to provide a list of recent projects that you can open with a single mouse click (these projects are also available through the File | Recent Projects submenu). Each project is identified with a last modified date.

On the left of the start page are some additional options that allow you to configure Visual Studio .NET, connect to the developer community, and find important MSDN documentation.

- **Get Started** is the default page with a list of projects. At the bottom of the page are two buttons that allow you to open a project (by browsing) or create a new one.

- **What's New** takes you to some of the MSDN documentation installed with Visual Studio .NET. It explains (in a brief overview) some of the fundamental differences between Visual Basic .NET and its earlier versions.

- **Online Community** provides a list of Visual Studio .NET newsgroups hosted by Microsoft on the Internet (see Figure 8-2). When you click on a newsgroup, your default news reader (typically Microsoft Outlook) will open to allow you to read and browse postings. Newsgroups are an excellent place to ask programming questions and communicate with other ASP.NET programmers.

Figure 8-2: The Online Community

- **Headlines** includes a wealth of information from Microsoft's MSDN developer site. You can use this section to read recent articles from the knowledge base (answers to technical problems or fixes and workarounds for known bugs), technical articles, and news releases. When you click on a link, the related web page will open inside Visual Studio .NET. If you want more comfortable browsing using your default Internet browser, just right-click on the link and choose External Window.

- **Search Online** allows you to search the MSDN online library (see Figure 8-3). This search option is available through the MSDN site (http://msdn.microsoft.com), but Visual Studio .NET makes it easy to search for information without needing to browse to a page or use an Internet user interface. This is an example of how you might want to use Web Services to integrate Internet features into desktop programs. For more information, refer to the fourth part of this book.

Figure 8-3: Searching the MSDN Online Library

- **Downloads** provides access to source code, service packs, and other updates. You'll also find add-ons beyond just program updates. For example, you might find tools for programming with Office applications or mobile (embedded) browsers.

- **XML Web Services** allows you to search for Web Services or register your own using the UDDI registry. Web Services are detailed in the fourth part of this book.

- **Web Hosting** is a special feature that lets you find companies that host ASP.NET sites and, in many cases, directly upload your ASP.NET pages from Visual Studio .NET to the remote site.

- **My Profile** provides configuration options that let you set the default language and help options, as well as the window layout and action on startup. Many more configuration options are provided in the Options window (select Tools | Options from the menu). You can also configure toolbars, menus, and create keyboard shortcuts from the Customize window (select Tools | Customize from the menu).

Creating a New Application

To start your first Visual Studio .NET application, click the New Project button on the start page or select File | New | Project.

The New Project window (shown in Figure 8-4) allows you to choose the type of project and the location where it will be created. You should choose the Visual Basic Projects node, and the ASP.NET Web Application template. You must then enter the virtual directory where you want to create the web application. For this to work, you must have already created the virtual directory using IIS Manager. The project will be created when you click OK.

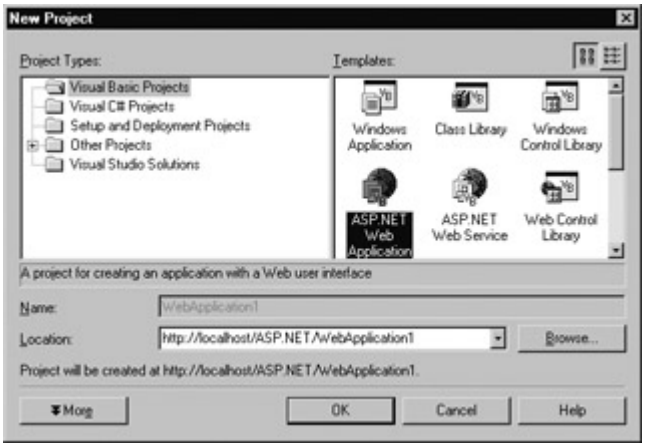


Figure 8-4: The New Project window

Interestingly, Visual Studio .NET can automatically create a project subdirectory in the virtual directory you chose. For example, if your computer has the virtual directory http://WebApps, you can create the application http://WebApps/Ecommerce in Visual Studio .NET. The project subdirectory (in this case Ecommerce) will automatically be configured as an application directory, which means the web pages will execute in a separate memory space from all other web applications.

Project Files

When you create a web application, Visual Studio .NET will add a collection of files. You can see all the files that your project contains in the Solution Explorer window (see Figure 8-5).

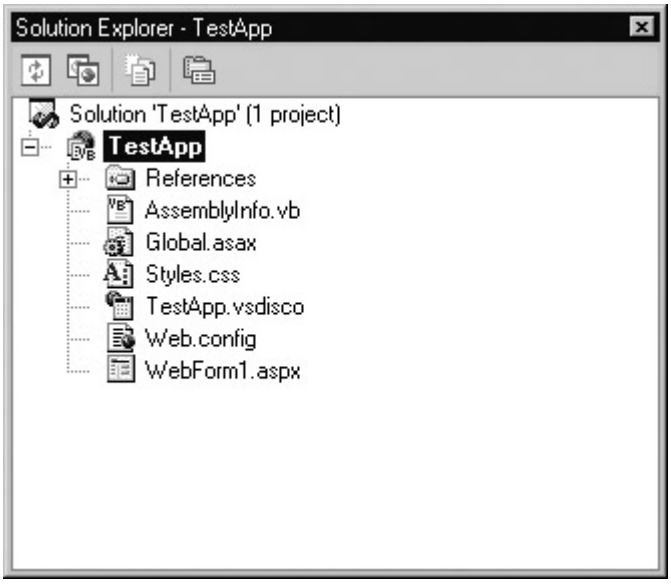


Figure 8-5: The Solution Explorer window

The Solution Explorer also shows all the references configured for your project. These are the .NET assemblies that your code-behind files link to automatically. If you want to use additional features, you may need to import more assemblies.

When creating files outside of Visual Studio .NET, you don't need to manually specify these references because they are resolved dynamically at runtime. However, if you want to precompile a code-behind file, you need to list all the referenced namespaces as parameters on the vbc.exe command line. Because Visual Studio .NET compiles all code-behind files, it needs to explicitly reference them.

The Solution Explorer also contains a file called AssemblyInfo.vb, which defines information that will be added to the compiled DLL file for your web application. This includes information such as the web application version number, company name, and so on. This information doesn't affect the performance of your web application, but it does make it easier to track multiple versions.

Assemblies Can Contain Multiple Namespaces

Once you create a reference to an assembly, you can use the types defined in that assembly. Assemblies can contain more than one namespace. For example, the System.Web.dll assembly includes classes in the System.Web namespace and the System.Web.UI namespace.

Adding a reference is not the same as using the Imports statement. The Imports statement creates an alias to a namespace in an available assembly. If you haven't added the reference, the namespace won't be available for importing. (For example, if you haven't imported the System.Web.dll, you won't be able to import the System.Web.UI namespace or use any of its classes.) Similarly, if you add a reference but don't import a namespace, you will have to use fully qualified names for classes in that namespace (for example, you'll need to write System.Web.UI.Page instead of just Page).

A sample AssemblyInfo.vb file is shown here:

```
<Assembly: AssemblyTitle("MyWebApp")>
<Assembly: AssemblyDescription("ProseTech Inc. E-Commerce Site.")>
<Assembly: AssemblyCompany("ProseTech")>
<Assembly: AssemblyCopyright("Copyright 2001")>
<Assembly: AssemblyTrademark("ProseTech™")>
```

Your project also includes a global.asax global application file, a web.config configuration file, a .vsdisco file used with Web Services, a Styles.css file you can use to create CSS styles, and all the .aspx files in your application.

Visual Studio .NET also creates a single .vbproj project file that references all these files. In addition, you can optionally combine more than one project into a .sln file. This technique is useful if you want to test a component with a web site, or Web Service project with a Windows application. It's described in Chapter 20.

The relationships of Visual Studio .NET project files are shown in Figure 8-6.

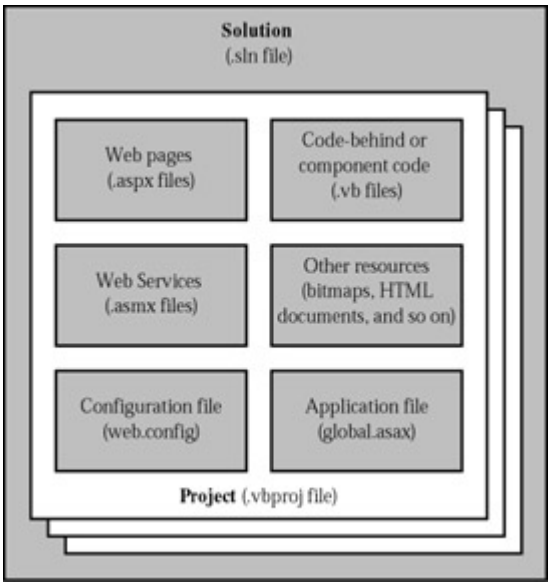


Figure 8-6: Project files

Configuring a File

To get information about a file, click on it once in the Solution Explorer. You can then read various pieces of information in the Properties window (see Figure 8-7). For example, the Build Action property describes what will happen when you run the application in Visual Studio .NET (typically, it will compile the file). The File Name property allows you to rename files without leaving the IDE.

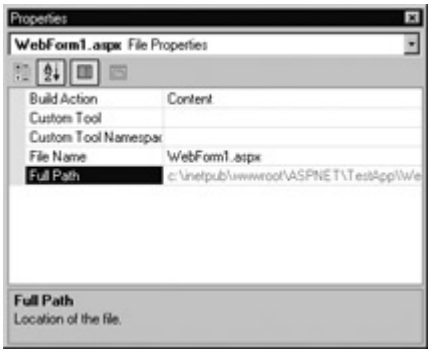


Figure 8-7: File properties

The Properties Window Is Grouped with Dynamic Help

By default, Visual Studio .NET groups some windows together in special tabs to save on-screen space. For example, the Properties window and the Dynamic Help window are both placed at the bottom of the window, and you can switch back and forth by clicking the appropriate tab. You can also close the Dynamic Help window entirely if you find it is slowing your computer down, or drag it and dock it to a different screen location. Just remember, if you end up with windows strewn all over the place and no easy way to get them back to their original configuration, select Tools |

Options from the menu, go to the Environment | General settings, and click Reset Window Layout to return to normal.

Adding a File

When you first create a web application, you will begin with one web page called WebForm1.aspx. You can add additional web pages by right-clicking on your project in the Solution Explorer, and selecting Add | Add New Item.

You can add various different types of files to your project, including web forms (the default), Web Services, standalone components, resources you want to track like bitmaps and text files, and even ordinary HTML and XML files (see Figure 8-8). Visual Studio .NET event provides some basic designers that allow you to edit these types of files directly in the IDE.

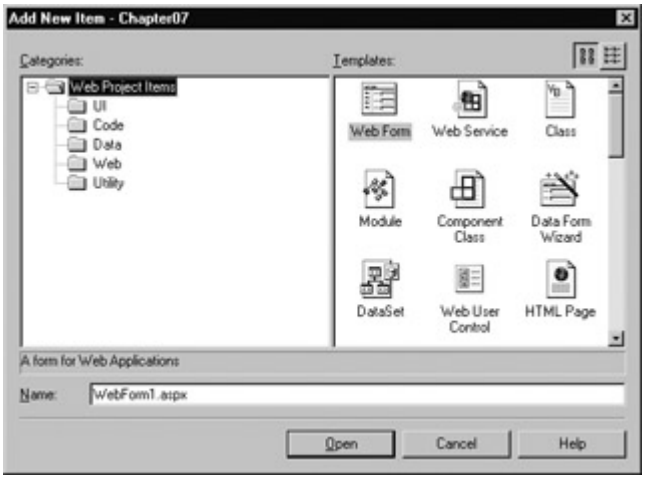


Figure 8-8: File types

The Web Form Designer

Now that you understand the basic organization of Visual Studio .NET, you can begin designing a simple web page. To start, double-click on the web page you want to design (start with WebPage1.aspx if you haven't added any additional pages). A special blank designer page will appear.

Before you begin, you may want to switch to grid layout, which gives you complete freedom to place controls wherever you want on the page. Select DOCUMENT in the Properties window (or just click on the blank web form once), and find the pageLayout property.

- In a FlowLayout page, elements are positioned line by line, like in a word processor document. To add a control, you need to drag and drop it to an appropriate place. You also need to add spaces and hard returns to position elements the way you want them.

- In a GridLayout page, elements can be positioned with absolute coordinates. You can draw controls directly onto the web form surface. The disadvantage is that if a control becomes much larger (for example, you place more text in a label at runtime) it could overwrite another adjacent control.

To add controls, choose the control type from the toolbox on the right. By default, the toolbox is enabled to automatically hide itself when your mouse moves away from it, somewhat like the AutoHide feature for the Windows taskbar. This behavior is often annoying, so you may want to click the pushpin in the top-right corner of the toolbox to make it stop in its fully expanded position.

You Can Combine Layout Modes

Although you must choose GridLayout or FlowLayout for the page, you can place controls inside special Grid Layout or Flow Layout panels. This allows you to align some controls relative to one another, but place them at a specific location on the page. (It also allows you to do the converse: align controls absolutely in a Grid Layout box, but have this box change position to accommodate the content on the rest of a FlowLayout page.)

You'll find the Grid Layout Panel and Flow Layout Panel in the HTML tab on the toolbox. These controls look like boxes. You can configure the border and background of a panel by right-clicking on it and choosing Build Style. These controls are actually just tags that use a special attribute to tell Visual Studio .NET how to treat them:

```
<div ms_positioning="FlowLayout">[HTML and controls go
here.]</div>
```

In the online examples, you'll find many cases that use this control.

There are two types of controls that you can add to web pages from the toolbox. Click on the Web Forms tab to add a web control. Click on the HTML tag to add an HTML element, which can be either a static tag or a server control. Once you have chosen the type of control, you can place it on your web form and resize it to the right dimensions (see Figure 8-9).

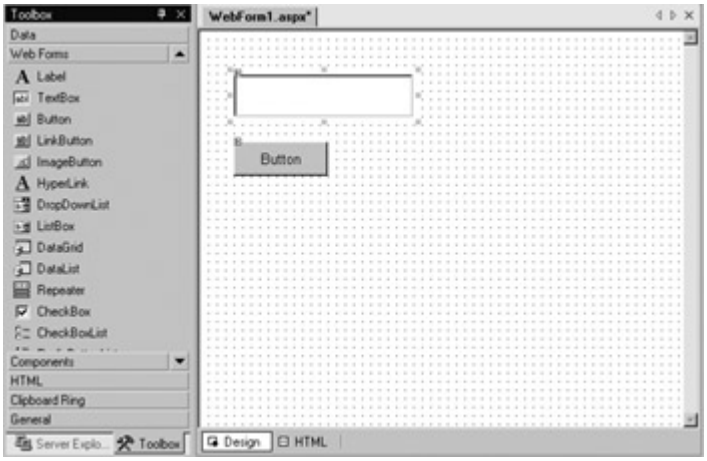


Figure 8-9: The web form designer

Every time you add a web control, Visual Studio automatically adds the corresponding tag to your .aspx file. You can even look at the .aspx code, or add server control tags and HTML elements manually by typing them in. To switch your view, click the HTML button at the bottom of the web designer. You can click Design to revert back to the graphical web form designer.

Figure 8-10 shows the HTML view of what you might see after you add a TextBox control to a web page.

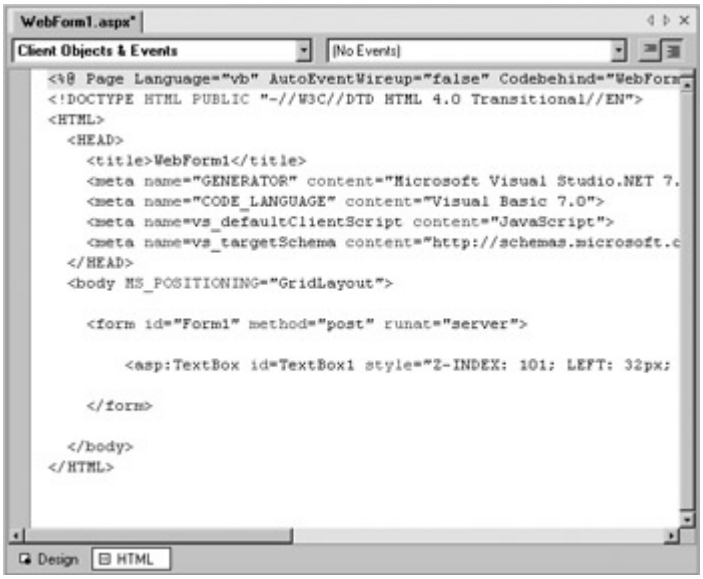


Figure 8-10: The HTML view

You can manually add attributes or rearrange controls. In fact, Visual Studio .NET even provides limited IntelliSense features that automatically complete opening tags and alert you if you use an invalid tag. Generally, however, you won't need to use the HTML view in Visual Studio .NET. Instead, you can switch back to the design view, and configure controls using the Properties window.

To configure a control, click once to select it (or choose it by name in the drop-down list at the top of the Properties window). Then modify the appropriate properties in the window, such as Text, ID, and ForeColor. These settings are automatically translated to the corresponding ASP.NET control tag attributes and define the initial appearance of your control. Visual Studio .NET even provides special "choosers" that allow you to select extended properties. For example, you can set a color from a drop-down list that shows you the color (see Figure 8-11), and you can configure the font from a standard font selection dialog box.

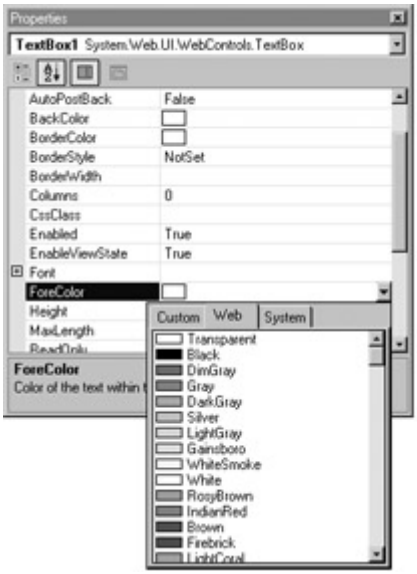


Figure 8-11: Control properties

If you try to configure an HTML control, you'll realize that the properties you are given correspond to the static HTML element, not the server control. Before you can use an HTML element as a server control, you need to right-click on it in the web page, and select Run As Server Control. This adds the required runat="server" attribute. (Alternatively, you could switch to design view and type this in on your own.)

Of course, not all HTML elements need to be server controls. For example, you might want to create a simple tag (provided as the Grid Layout panel or Flow Layout panel). Visual Studio .NET still provides you with some indispensable tools to create this basic tag. For example, you can right-click on the panel and choose Build Style. The Style Builder window will appear with "single-stop shopping" for all the style attributes you need to use (see Figure 8-12). As you configure the text, color, and border properties, the control will be updated on the web page to reflect your settings.

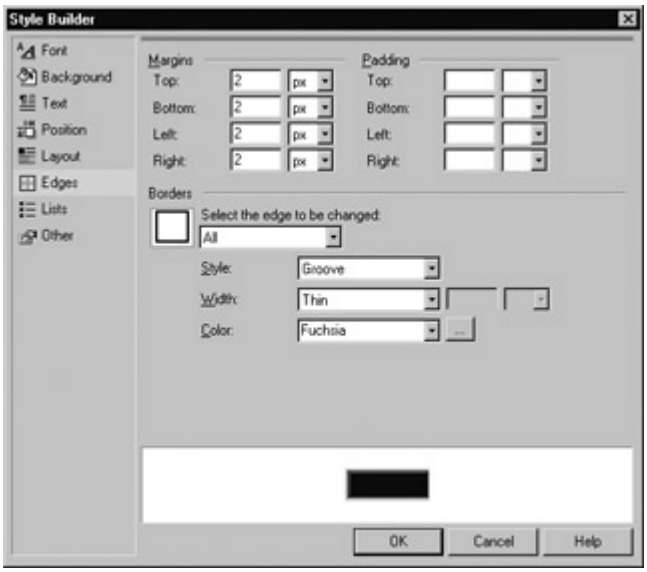


Figure 8-12: Building HTML styles

You'll also notice that the DOCUMENT object (representing the web page) allows you to configure many additional web page options. For example, if you set enableSessionState to False, the Page directive in the .aspx file will automatically include the attribute EnableSessionState=False.

```
<%@ Page Language="vb" AutoEventWireup="false"
    Codebehind="WebForm1.aspx.vb"
    Inherits="TestProject.WebForm1"%>
```

The Page directive also hints at a couple of other interesting points. The Codebehind attribute indicates that Visual Studio .NET's standard is to create a code-behind file with the same name as the .aspx file, but with the extension .aspx.vb. Our examples so far have used a slightly different convention, and given code-behind files the same name with the extension .vb (as in WebForm1.vb instead of WebForm1.aspx.vb).

You'll also notice that the Inherits directive references a class with the same name in a namespace defined by your project. In other words, when you create a project called Test and a web page class called TestPage, it will be provided as Test.TestPage. This default namespace can be configured using the project options, as you'll see later in this chapter.

Writing Code

Some of Visual Studio .NET's most welcome enhancements appear when you start to write the code that supports your user interface. To start coding, you need to switch to the code-behind view. To switch back and forth, you can use two buttons that are placed just above the Solution Explorer window as shown in the following illustration. The ToolTips read View Code and View Designer. The "code" in question is the VB code, not the .aspx markup.

When you switch to code view, you'll see that Visual Basic automatically creates a Page class for you as you design it. It also adds the appropriate member variable, with the appropriate name, for each control you have added. This saves many extra keystrokes.

Visual Studio .NET also adds a special block of extra code into a collapsible region titled Web Form Designer Generated Code. This code, shown below, simply initializes the page so it will work with Visual Studio .NET in the design environment, and can safely be ignored.

```
#Region " Web Form Designer Generated Code "

    'This call is required by the Web Form Designer.
    <System.Diagnostics.DebuggerStepThrough()>_
    Private Sub InitializeComponent()
    End Sub

    Private Sub Page_Init(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Init
        'CODEGEN: This method call is required by the Web Form Designer
        'Do not modify it using the code editor.
        InitializeComponent()
    End Sub

#End Region
```

Where Are the Code-Behind Files?

You may have noticed that the code-behind files don't appear in the project list. In fact, they are there, but they are hidden by default to present a simpler programming model to new users. This gives the illusion that the .aspx file and the .vb file are combined in one entity.

To see a code-behind file, select Project | Show All Files. Now the Solution Explorer will show an .aspx.vb code-behind file under each .aspx page.

Adding an Event

There are three ways to add an event to your code:

Type it in manually You must specify the appropriate parameters and the Handles clause, just like when we created the files by hand in the previous two chapters.

Double-click on a control in the design view Visual Studio .NET will create an event handler for that control's "default event." For example, if you double-click on the page, it will create a Page.Load event handler. If you double-click on a button or input control, it will create the required click or change event handler.

Choose the event from the drop-down list At the top of the code view, there are two drop-down lists. To add an event handler, choose the control in the list on the left, and the event you want to react to in the list on the right (see Figure 8-13). To add an event for the page, choose MyBase for the control name.

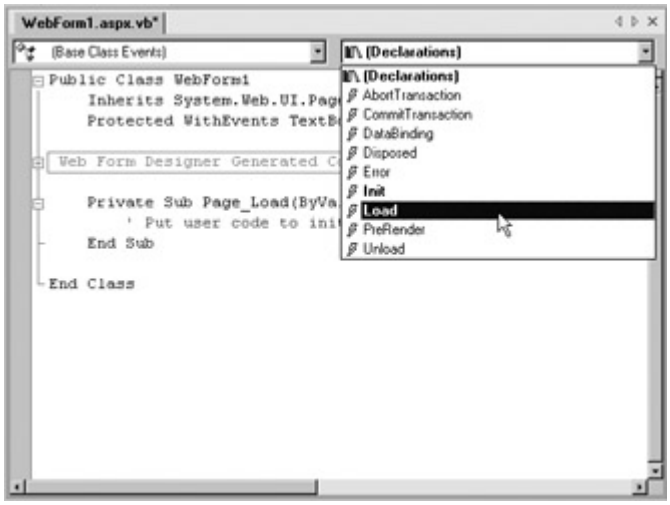


Figure 8-13: Creating an event handler

When Visual Studio .NET creates an event handler for you, it automatically adds the required Handles clause. It also adds ByVal before each parameter. This indicates that you receive a copy of the parameter, and that any modifications you make will not affect the calling code (in this case, the .NET framework). This keyword is purely optional—in fact, in the examples in this book it's left out to save space.

IntelliSense and Outlining

Visual Studio .NET provides a number of automatic timesavers through its IntelliSense technology. They are similar to features such as automatic spell checking and formatting in Microsoft Office applications, but much less intrusive. Most of these features are introduced in the following sections, but you'll need several hours of programming before you've become familiar with all of Visual Studio .NET's timesavers. There is not enough space to described advanced tricks, such as special edit commands that comment entire blocks of text, intelligent search and replace, and programmable macros—these features could occupy an entire book of their own.

Outlining

Outlining allows Visual Studio .NET to "collapse" a subroutine, block structure, or region to a single line. It allows you to see the code that interests you, while hiding unimportant code. To collapse a portion of code, click the minus box next to the first line. Click the box again (which will now have a plus symbol) to expand it (see Figure 8-14).

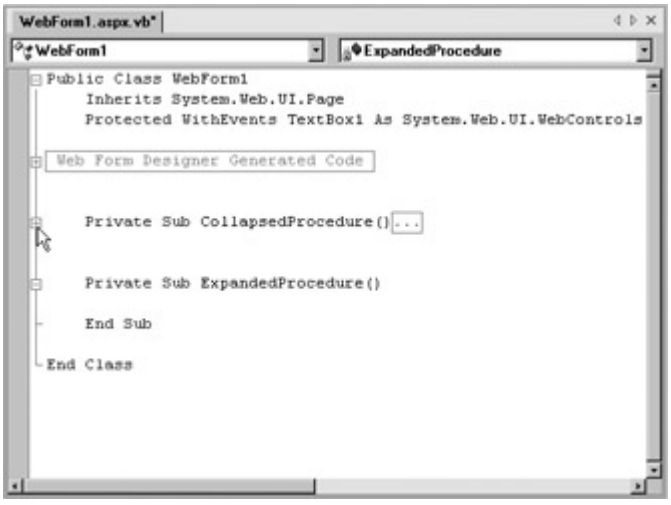


Figure 8-14: Collapsing code

Member List

Visual Studio .NET makes it easy for you to interact with controls and classes. When you type a class or object name, it pops up a list of available properties and methods (see Figure 8-15). It uses a similar trick to provide a list of data types when you define a variable, or to provide a list of valid values when you assign an enumeration.

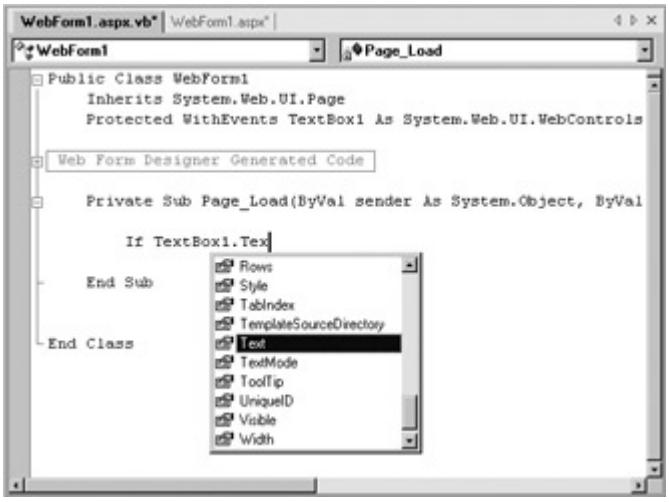


Figure 8-15: IntelliSense at work

Visual Studio .NET also automatically provides a list of parameters and their data types when you call a method or invoke a constructor. This information is presented in a ToolTip above the code and is shown as you type. Because the .NET class library makes heavy use of function overloading, these methods may have multiple different versions. When they do, Visual Studio .NET indicates the number of versions, and allows you to see the method definitions for each one by clicking the small up and down arrows in the ToolTip. Each time you click the arrow, the ToolTip displays a different version of the overloaded method (see Figure 8-16).

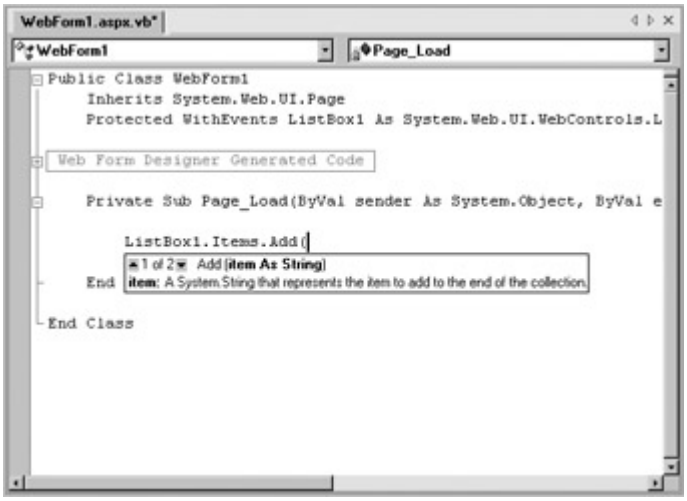


Figure 8-16: IntelliSense with an overloaded method

Block Completion

You'll also save a few keystrokes when typing in common code constructs. For example, Visual Studio .NET will automatically add the End If, End Case, or Next to the end of your code when needed.

Error Underlining

One of the code editor's most useful features is error underlining. Visual Studio .NET is able to detect a variety of error conditions, such as undefined variables, properties or methods, invalid data type conversions, and missing code elements. Rather than stopping you to alert you that a problem exists (an annoying feature in previous versions of Visual Basic), the Visual Studio .NET

editor quietly underlines the offending code. You can hover your mouse over an underlined error to see a brief ToolTip description of the problem (see Figure 8-17).

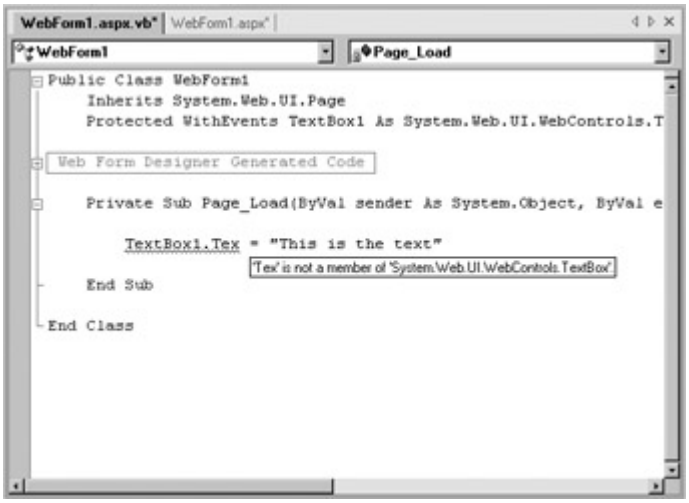


Figure 8-17: IntelliSense with an error

If you try to compile your program with errors in it, Visual Basic .NET stops you and displays the Task List window with a list of all the errors it detected (see Figure 8-18). You can then jump quickly to a problem by double-clicking it in the list.

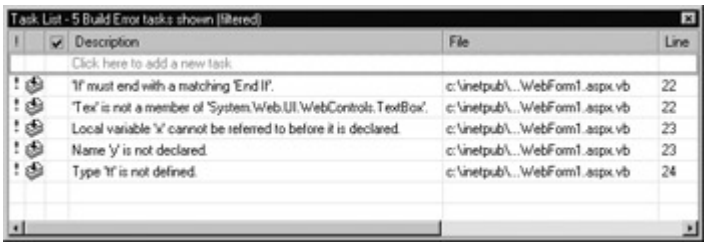


Figure 8-18: Build errors in the Task List

Auto Format and Color

Visual Studio .NET also provides some cosmetic conveniences. It automatically colors your code, making comments green, keywords blue, and normal code black (although you can configure these options). The result is much more readable code.

In addition, Visual Studio .NET is configured by default to automatically format your code. This means you can type your code lines freely without worrying about tabs and positioning. As soon as you move on to the next line, Visual Studio .NET applies the "correct" indenting.

Project Settings

There are several types of project-wide settings that you can configure. One of the most important is setting your start page. The start page isn't necessarily the page that users will start with (that depends on the URL they use, or it will default to the index.html file if they type in a virtual directory name with no file specified). Instead, the start page is the page that Visual Studio .NET will launch automatically when you are testing and running your application. To set a start page, right-click on the page in the Solution Explorer and select Set As Start Page.

To access the full set of project properties, right-click on your project in the Solution Explorer and select Properties (or select Project | Properties from the menu). The window shown in Figure 8-19 will appear. There are two groups of project settings: Common Properties, which always apply, and Configuration Properties, which are tied to a specific type mode (for example, release mode or debug mode). The Common Properties are explained in Table 8-1.

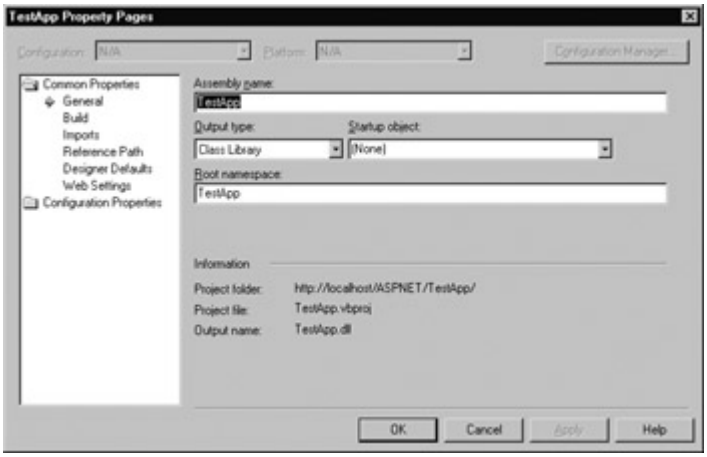


Figure 8-19: Project properties

Table 8-1: Common Project Properties

Group	Description
General	Configure the Assembly name setting to change the name of the DLL file that Visual Studio .NET generates. Change the Root namespace to change the name of the namespace that contains all your web classes (the page directive in the .aspx files will be updated automatically). Note that the Output type will always be class library. A web application is not a standalone executable, but a DLL that the ASP.NET engine loads in response to a user request.
Build	Allows you to set the project-wide values for Option Explicit, Option Strict, and Option Compare. Option Explicit should always be enabled to prevent variable naming mistakes. Option Strict forces you to use explicit data type conversion (for example, you must use the CType function to convert a string into a number), which can help eliminate accidental errors.
Imports	You can use this section to import namespaces and make them locally available to all the code in your project. When you take this approach, you don't need to use the Imports statement at the beginning of each file.
Designer Defaults	You can set the default page layout style (grid or layout) that will be used in all new web pages. You can also specify that client event handling and postback code should be created using VBScript. Generally, you won't use this option as it limits your application to Internet Explorer browsers.

Visual Studio .NET Debugging

Once you have created an application, you can start it by choosing Debug | Start or clicking the Start button in the toolbar. When you compile your project in debug mode, Visual Studio .NET adds special debugging symbols to your code, which allow it to work with the built-in debugging services. When you are ready to deploy your web application, you change the build configuration in the toolbar to Release (see the following illustration), which instructs Visual Studio .NET to compile your program without debug symbols.

When you compile your application, Visual Studio .NET creates a single DLL file with the name of

your project and compiles all the page classes into this file. (This process was described in the explanation of code-behind development in Chapter 5.) Visual Studio .NET then starts one of the pages in your application by launching Internet Explorer and browsing to the .aspx file.

Single-Step Debugging

Single-step debugging allows you to test your assumptions about how your code works, and see what is really happening under the hood of your application. It's incredibly easy to use:

1. Find a location in your code where you want to pause execution, and start single-stepping (you can use any executable line of code, but not a variable declaration, comment, or blank line). Click in the margin next to the line code, and a red breakpoint will appear (see Figure 8-20).

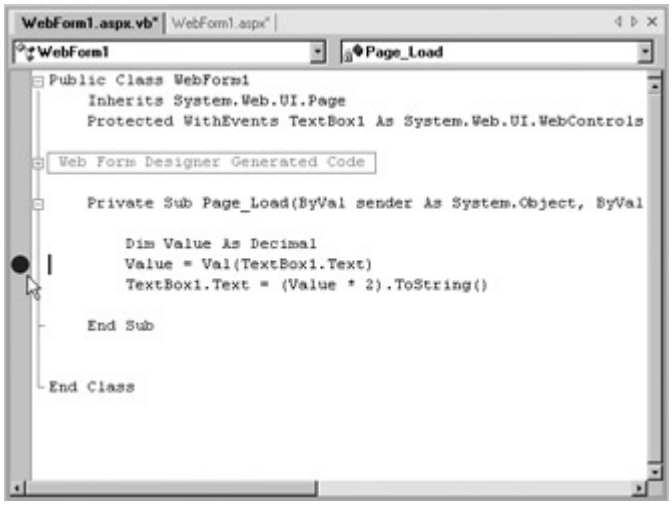


Figure 8-20: Setting a breakpoint

2. Now start your program as you would ordinarily. When the program reaches your breakpoint, execution will pause, and you'll be switched back to the Visual Studio .NET code window. The breakpoint statement won't be executed.

3. At this point, you can execute the current line by pressing f8. The following line in your code will be highlighted with a yellow arrow, indicating that this is the next line that will be executed. You can continue like this through your program, running one line at a time by pressing f8, and following the code's path of execution.

4. Whenever the code is in break mode, you can hover over variables to see their current contents (see Figure 8-21). This allows you to verify that variables contain the values you expect.

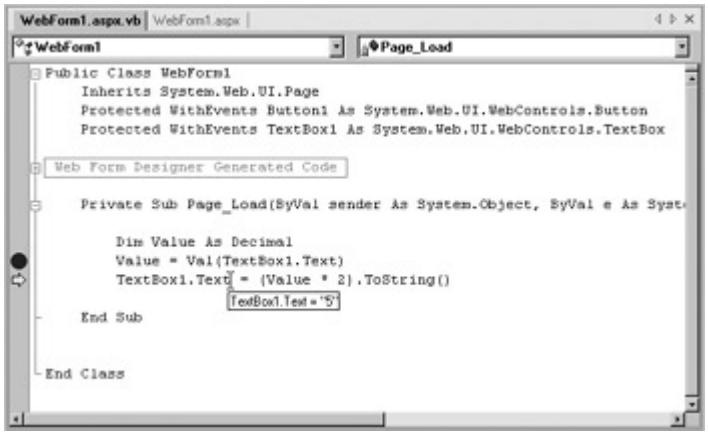


Figure 8-21: Viewing variable contents in break mode

You can also use any of the commands listed in Table 8-2 while in break mode. These commands are available from the context menu by right-clicking on the code window, or by using the associated hotkey.

Breakpoints are automatically saved with your application, although they aren't used when you compile the application in release mode. You can also create and then disable a breakpoint. Just click on an active breakpoint, and it will become a transparent gray circle. That allows you to keep a breakpoint to use later, without leaving it active.

Table 8-2: Commands Available in Break Mode

Command (Hotkey)	Description
Step Into (F8)	Executes the currently highlighted line and then pauses. If the currently highlighted line calls a procedure, execution will pause at the first executable line inside the method or function (which is why this feature is called stepping "into").
Step Over (SHIFT-F8)	The same as Step Into, except it runs procedures as though they are a single line. If you press Step Over while a procedure call is highlighted, the entire procedure will be executed. Execution will pause at the next executable statement in the current procedure.
Step Out (CTRL-SHIFT-F8)	Executes all the code in the current procedure, and then pauses at the statement that immediately follows the one that called this method or function. In other words, this allows you to step "out" of the current procedure in one large jump.
Continue (F5)	Resumes the program and continues to run it normally, without pausing until another breakpoint is reached.
Run To Cursor (CTRL-F8)	Allows you to run all the code up to a specific line (where your cursor is currently positioned). You can use this technique to skip a time-consuming loop.
Set Next Statement (CTRL-F9)	Allows you to change the path of execution of your program while debugging. It causes your program to mark the current line (where your cursor is positioned) as the current line for execution. When you resume execution, this line will be executed, and the program will continue from that point.
Show Next Statement	Moves the focus to the line of code that is marked for execution. This line is marked by a yellow arrow. The Show Next Statement command is useful if you lose your place while editing.

Choose Debug | Windows | Breakpoints to see a window that lists all the breakpoints in your current project. The Breakpoints window provides a hit count, showing the number of times a breakpoint has been encountered (see Figure 8-22). You can jump to the corresponding location in code by double-clicking on a breakpoint.

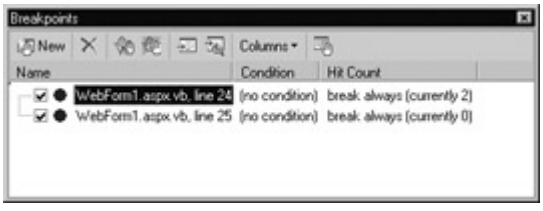


Figure 8-22: Breakpoints window

Advanced Breakpoints

Visual Studio .NET allows you to customize breakpoints so they only occur if certain conditions are true. To customize a breakpoint, right-click on it and select Breakpoint Properties. In the window that appears:

- Click the Condition button to set an expression. You can choose to break when this expression is True, or when it has changed since the last time the breakpoint was hit.
- Click the Hit Count button to create a breakpoint that only pauses after a breakpoint has been hit a certain number of times (for example, at least 20), or a specific multiple of times (for example, every fifth time).

Variable Watches

What if you want to track a variable, but you don't want to stop at a specific point? You can switch your program into break mode at any point by clicking the Pause button in the toolbar or selecting Debug | Break All. You can also track variables across an entire application using the Autos, Locals, and Watch windows, described in Table 8-3.

Table 8-3: Variable Watch Windows

Window	Description
Locals	Automatically displays all the variables that are in scope in the current procedure. This offers a quick summary of important variables.
Autos	Automatically displays variables that Visual Studio .NET determines are important for the current code statement. For example, this might include variables that are accessed or changed in the previous line.
Watch	Displays variables you have added. Watches are saved with your project, so you can continue tracking a variable at a later time. To add a watch, right-click on a variable in your code and select Add Watch, or double-click on the last row in the Watch window and type in the variable name.

Each row in the Locals, Autos, and Watch windows provides information about the type or class of the variable and its current value. If the variable holds an object instance, you can expand the variable and see its private members and properties. For example, in the Locals window you'll see the variable Me, which is a reference to the current page class. If you click on the plus (+) box next to the word Me, a full list will appear that describes many page properties (and some system values), as shown in Figure 8-23.

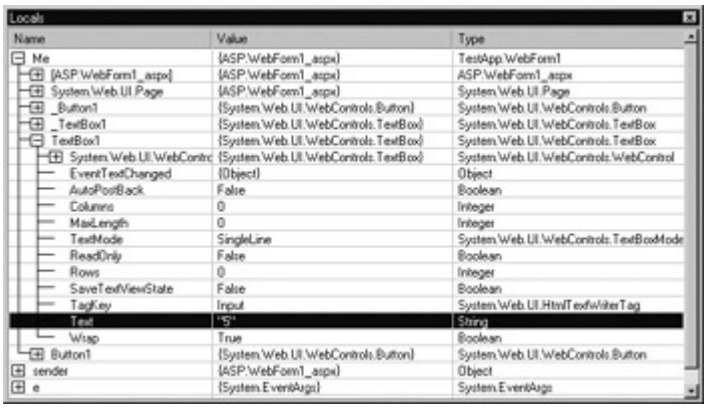


Figure 8-23: Viewing the page class in the Locals window

If you are missing one of the Watch windows, you can show it manually by selecting it from the

Debug | Windows submenu.

You Can Modify Variables in Break Mode

The Watch, Locals, and Autos windows allow you to change simple variables while your program is in break mode. Just double-click on the current value in the Value column and type in a new value. This allows you to simulate scenarios that are difficult or time-consuming to recreate manually, or test specific error conditions.

Working Without Visual Studio .NET

This chapter has extolled the many benefits of Visual Studio .NET. However, if you prefer to work with a simpler, leaner utility (such as Notepad), you can still create equally advanced ASP.NET applications. You might even discover a third-party IDE that you prefer.

The examples with this book are compatible with Visual Studio .NET or with any other type of code-behind development. The online code has standard .aspx and .vb files that you can use independently or use with Visual Studio .NET by using the .vbproj and .sln project and solution files.

Chapter 9: Validation and Rich Controls

Overview

The ASP.NET web control framework has almost unlimited possibilities. In the coming months, we'll see third-party component developers create increasingly sophisticated control classes that you can plug in to your web applications effortlessly. In this chapter, we start to look at some of the real promise of ASP.NET and the server-control model, and consider controls that have no equivalent in the ordinary HTML world: the Calendar and AdRotator.

The second part of this chapter examines ASP.NET's validation controls. These controls take a previously time-consuming and complicated task—verifying user input and reporting errors—and automate it with an elegant, easy-to-use collection of validators. You'll learn how to add these controls to an existing page, and use regular expression, custom validation functions, and manual validation. And as usual, we'll peer under the hood to take a look at how ASP.NET implements these new features.

Finally, we'll briefly consider the next generation of controls, which promises to bring a new world of rich user interface to the Web. These new controls, prepared by other developers, represent one of the most exciting new directions for ASP.NET.

The Calendar Control

The Calendar control is one of the most impressive web controls. It's commonly called a rich control because it can be programmed as a single object (and defined in a single, simple tag), but rendered in dozens of lines of HTML output.

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

In its default mode, the Calendar control presents a single month view (see Figure 9-1) where the user can navigate from month to month using the navigational areas, at which point the page is posted back, and ASP.NET automatically provides a new page with the correct month values. When the user clicks on a date, it becomes highlighted in a gray box. You can retrieve the day as a DateTime object from the Calendar.SelectedDate property.

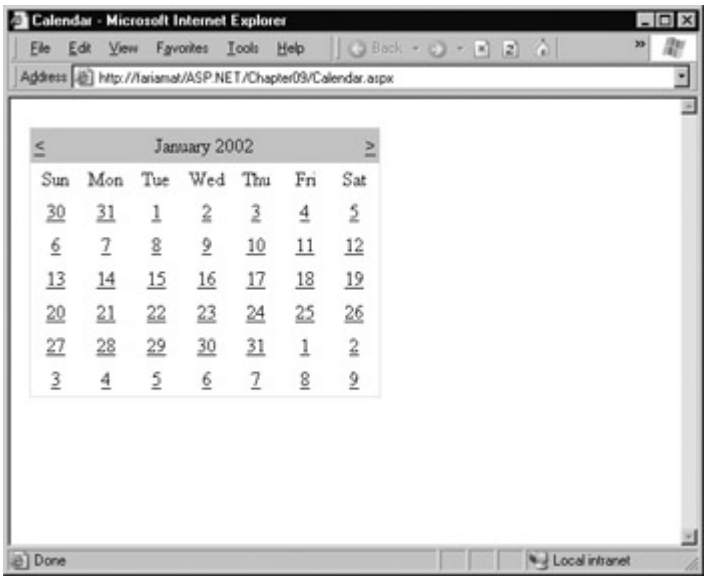


Figure 9-1: The default Calendar

This basic set of features may provide everything you need in your application. Alternatively, you can configure different selection modes to allow users to select entire weeks or months, or render

the control as a static calendar that doesn't allow selection. The important fact to 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.

The type of selection is set through the Calendar.SelectionMode property. You may also need to set the Calendar.FirstDayOfWeek property to configure how a week is selected. (For example, set FirstDayOfWeek to the enumerated value Monday, and weeks will be selected from Monday to Sunday.)

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 using the For Each syntax. The following code demonstrates this technique (see Figure 9-2).

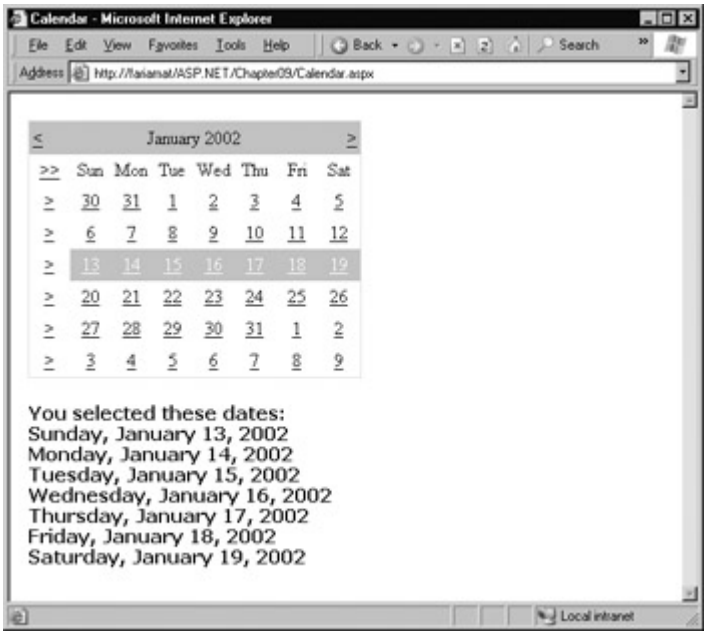


Figure 9-2: Selecting multiple dates

```
lblDates.Text = "You selected these dates:<br>"

Dim dt As DateTime
For Each dt In MyCalendar.SelectedDates
   lblDates.Text &= dt.ToLongDateString() & "<br>"
Next
```

Formatting the Calendar

The Calendar control provides a whole host of formatting-related properties. These are described in more detail in Chapter 27. It's enough to note that various parts of the Calendar, like the header, selector, and various day types can be set 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.

If you are using an IDE such as Visual Studio .NET, you can even set an entire related color scheme using the built-in designer. Simply right-click on the control on your design page, and select Auto Format. You will be presented with a list of predefined formats that set the style properties, as shown in Figure 9-3.

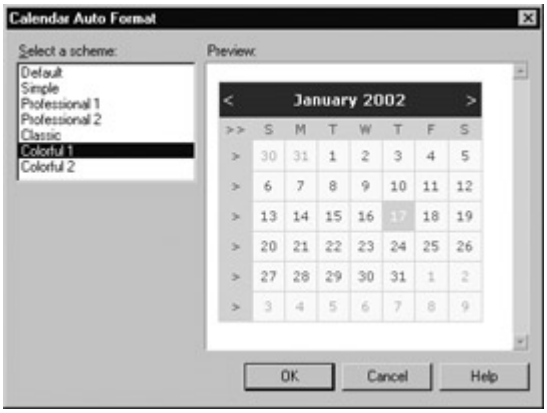


Figure 9-3: Calendar styles

You can also use additional properties to hide some elements or configure the text they display.

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 only provided on set days. The Calendar 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 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.

```
Private Sub DayRender(source As Object, e As DayRenderEventArgs) _
  Handles Calendar.DayRender

    ' Restrict dates after the year 2100, and those on the weekend.
    If e.Day.IsWeekend Or e.Day.Date.Year > 2100 Then
        e.Day.IsSelectable = False
    End If

End Sub
```

The e.Day object is an instance of the CalendarDay class, which provides various useful properties. These are described in Table 9-1.

Table 9-1: CalendarDay Properties

Property	Description
Date	The DateTime object that represents this date.
IsWeekend	True if this date falls on a Saturday or Sunday.
IsToday	True if this value matches the Calendar.TodaysDate property, which is set to the current day by default.
IsOtherMonth	True if this date does not 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 really a sophisticated HTML table). For example, you could highlight an important date or even add extra information (see Figure 9-4).

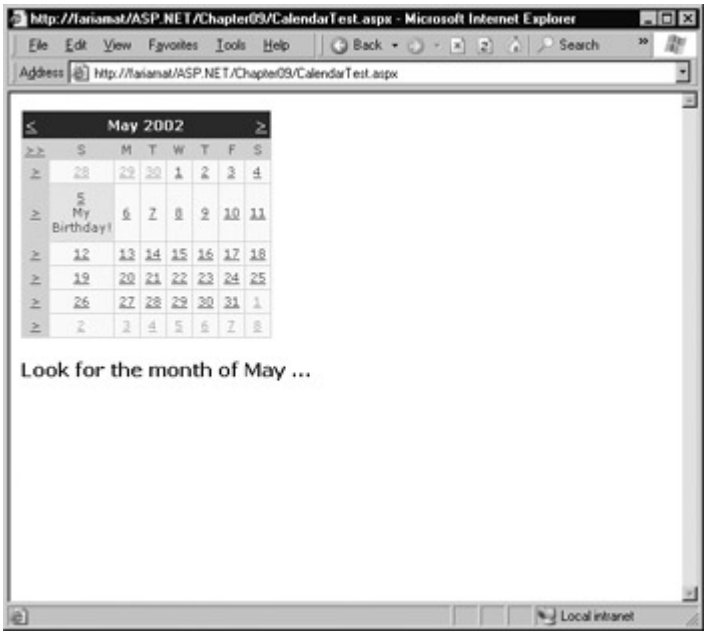


Figure 9-4: Highlighting a day

```
Private Sub DayRender(source As Object, e As DayRenderEventArgs) _
  Handles Calendar.DayRender

    ' Check for May 5 in any year, and format it.
    If e.Day.Date.Day = 5 And e.Day.Date.Month = 5 Then
        e.Cell.BackColor = System.Drawing.Color.Yellow

        ' Add some static text to the cell.
        Dim lbl As New Label
        lbl.Text = "<br>My Birthday!"
        e.Cell.Controls.Add(lbl)
    End If

End Sub
```

The Calendar control provides two other useful events: SelectionChanged and VisibleMonthChanged. These occur immediately after a change, but before the page is returned to the user. You can react to this event and update other portions of the web page to correspond to the current calendar month (for example, setting a list of valid times in a list control).

```
Private Sub SelectionChanged(source As Object, e As EventArgs) _
  Handles Calendar.SelectionChanged

    lstTimes.Items.Clear

    Select Case Calendar.SelectedDate.DayOfWeek
        Case DayOfWeek.Monday
            ' Apply special Monday schedule.
            lstTimes.Items.Add("10:00")
            lstTimes.Items.Add("10:30")
            lstTimes.Items.Add("11:00")
        Case Else
            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")
    End Select
End Sub
```

See These Features in Action

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

The AdRotator

The AdRotator has been available as an ASP component for some time. The new ASP.NET AdRotator adds some new features, such as the ability to filter the full list of banners to the best matches for a given page. The AdRotator also uses a new XML file format.

The basic purpose of the AdRotator is to provide a banner-type graphic on a page (often used as an advertisement link to another site) that is chosen randomly from a group of possible banners. In other words, every time the page is requested, a different banner could be chosen and displayed, which is the "rotation" indicated by the name AdRotator.

In 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 banner you would either have to repeat the code or create your own custom control. The AdRotator provides these features for free.

The Advertisement File

The AdRotator stores its list of image files in a special 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>
```

This example shows a single possible advertisement. 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 9-2.

Table 9-2: 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 on the banner.
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 as the banner with the value 5.
Keyword	A keyword that identifies a group of advertisements. This can be used 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 only provides a limited set of properties. You specify the appropriate advertisement file in the AdvertisementFile property, and the type of window that the link should follow (the Target window). Note that in Visual Studio .NET, you can't link to an advertisement file unless you have added it to the current project.

The target can name a specific frame, or it can use one of the special values defined in Table 9-3.

Table 9-3: 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 site appears in the full, unframed window).

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

```
<asp:AdRotator id="Ads" runat="server" AvertisementFile="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 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 9-5.

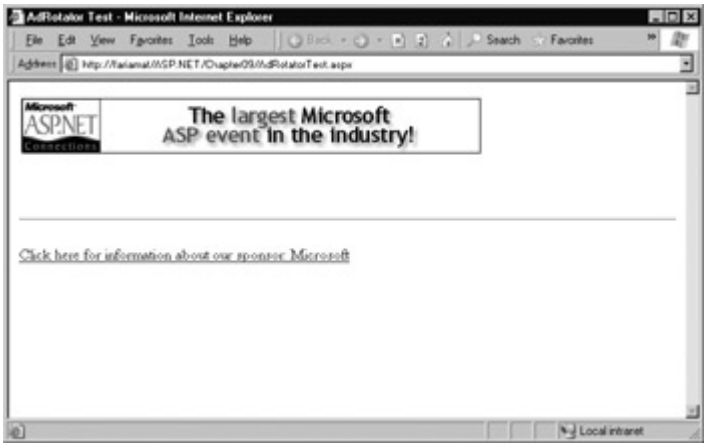


Figure 9-5: An AdRotator with synchronized content

The event-handling code for this example simply configures a hyperlink control.

```
Private Sub Ads_AdCreated(sender As Object, _
  e As AdCreatedEventArgs) Handles Ads.AdCreated
    ' Synchronize the Hyperlink control.
    lnkBanner.NavigateUrl = e.NavigateUrl

    ' Syncrhonize the text of the link.
    lnkBanner.Text = "Click here for information about our sponsor: "
    lnkBanner.Text &= e.AlternateText()
End Sub
```

As you can see, rich controls like 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.

Validation

As a seasoned developer, you probably realize that you can't assume users won't make mistakes. What's particularly daunting is the range of possible mistakes:

- Users might ignore an important field and leave it blank.
- Users might try to type a short string of "nonsense" to circumvent a required field check, creating endless headaches on your end, such as invalid email addresses that cause problems for your automatic mailing programs.
- Users might make an honest mistake, including a typing error, entering a non-numeric character in a number field, or submitting the wrong type of information. They 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 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 if the current character is valid, and prevent it from appearing if it isn't. This technique is commonly used to create text boxes that only accept numeric input.

In web applications, however, you don't have this sort of fine-grained control. In order to handle a KeyPress event the page would have to be posted back to the server every time the user types a letter, slowing an application down hopelessly. Instead, 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 web sites only report the first incorrect field, while others use a special table, list, or window that describes them all. By the time you have perfected your validation routines, a considerable amount of effort has gone into writing validation code.

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

The Validation Controls

ASP.NET provides five different validator controls (see Table 9-4). Four are targeted at specific types of validation, while the fifth allows you to apply custom validation routines.

Table 9-4: Validator Controls

Control Class	Description
RequiredFieldValidator	Validation succeeds as long as the input control does not 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, specified input control.
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.

Like all other web controls, you add a validator as a tag in the form <asp:ControlClassName />. There is one additional validation control, called ValidationSummary, that doesn't perform any actual control checking. Instead, it can be used to provide a list of all the validation errors for the entire page.

Validation Always Succeeds for Empty Values

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 is not the behavior you want, you should add an additional RequiredFieldValidator to the control. That ensures that two types of validation will be performed, effectively restricting blank values.

The Validation Process

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

1. The user receives a normal page, and begins to fill in the input controls.

2. When finished, the user clicks a button to submit the page.

3. Every Button control has a CausesValidation property.
    - If this property is False, ASP.NET will ignore the validation controls, the page will be posted back, and your event handling code will run normally.

    - If this property 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 that you will have to specifically check in the event handler whether the page is valid or not.

Based on this description, you'll realize that validation happens automatically when certain buttons are clicked. It does not happen when the page is posted back due to a change event (like 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. We'll look at this process in more detail a little later.

The Validator Classes

The validation control classes 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. Its properties are described in Table 9-5.

Table 9-5: 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.
ErrorMessage, ForeColor, and Display	If validation fails, the validator control can display a text message (set by the ErrorMessage property). The Display property allows you to configure whether this error message will be added dynamically as needed (Dynamic), or whether an appropriate space will be reserved for the message (Static). 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.
IsValid	After validation is performed, this returns True or False depending on whether it succeeded or failed. Generally, you will 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.
EnableClientSideScript	If set to True, ASP.NET will add special JavaScript and DHTML code to allow client-side validation on browsers that support it.

ASP.NET Also Provides Client-Side Validation

In browsers that support it (currently only Internet Explorer 5 and above), ASP.NET will automatically add code for client-side validation. In this case, when the user clicks on 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 the application.

However, if the page validates successfully on the client side, ASP.NET will still revalidate it when it is received at the server. This is because it is easy for an experienced user to circumvent client-side validation (for example, by deleting the block of JavaScript validation code). By performing the validation at both ends, your application can be as responsive as possible, but also secure.

When using a validation control, the only properties you need to use are ControlToValidate and ErrorMessage. In addition, you may need to use the properties that are used for your specific validator. These properties are outlined in Table 9-6.

Table 9-6: Validator-Specific Properties

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

A detailed explanation for each validation control is available in Chapter 27. Also, the customer validation form example later in this chapter demonstrates each type of validation.

A Simple Validation Example

To get an understanding of how validation works, you can create a simple example. This test uses a single Button web control, two TextBox controls, and a RangeValidation control that validates the first text box. If validation fails, an error message will be shown in the RangeValidation control (see Figure 9-6), so this control should be placed immediately next to the TextBox it's validating.

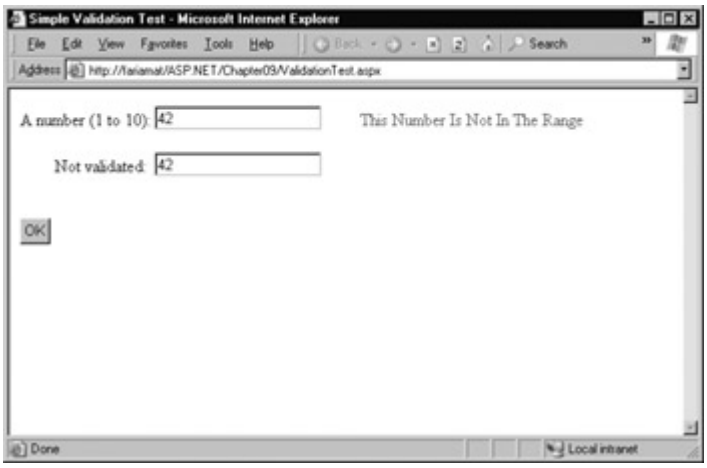


Figure 9-6: Failed validation

In addition, a Label control is placed at the bottom of the form. This label will report when the page has been successfully posted back and the button click event handling code is executed. Its EnableViewState property is disabled to ensure that it will be cleared every time the page is posted back.

The layout code defines a RangeValidator control, sets the error message, identifies the control that will be validated, and requires an integer from 1 to 10 (shown in bold in the following code

UNIT - 4
ADO.NET Data Access

Chapter 13: ADO.NET Data Access

Overview

Chapter 12 introduced ADO.NET and the family of objects that provides its functionality. This family includes objects for modeling data (representing tables, rows, columns, and relations) and objects designed for communicating with a data source. In this chapter, you'll learn how to put these objects to work by creating pages that use ADO.NET to retrieve information from a database and apply changes. You'll also learn how to retrieve and manage disconnected data, and deal with the potential problems that can occur.

About the ADO.NET Examples

One of the goals of ADO.NET is to provide data access universally and with a consistent object model. That means you should be able to access data the same way regardless of what type of database you are using (or even if you are using a data source that doesn't correspond to a database). In this chapter, most examples will use the "lowest common denominator" of data access—the OLE DB data types. These types, found in the System.Data.OleDb namespace, allow you to communicate with just about any database that has a matching OLE DB provider, including SQL Server. However, when you develop a production-level application, you'll probably want to use an authentic .NET provider for best performance. These managed providers will work almost exactly the same, but their Connection and Command objects will have different names and will be located in a different namespace. Currently, the only managed provider included with .NET is SQL Server, but many other varieties are planned, and a managed ODBC driver is already available as a download from Microsoft.

In the examples in this chapter, I make note of any differences between the OLE DB and SQL providers. Remember, the underlying technical details differ, but the objects are almost identical. The only real differences are as follows:

- The names of the Connection, Command, and DataReader classes are different, to help you distinguish them.
- The connection string (the information you use to connect to the database) differs depending on the way you connect and several other factors.
- The OLE DB objects support a wider variety of data types, while the SQL Server objects only support valid SQL Server data types.

Know Your Database

It's best not to worry too much about the differences between OLE DB and SQL providers. They are similar to the differences between different OLE DB data sources, or even between differently configured installations of the same relational database product. When programming with ADO.NET, it always helps to know your database. If you have information on-hand about the data types it uses, the stored procedures it provides, and the user login you require, you'll be able to work more quickly and with less chance of error.

Obtaining the Sample Database

This chapter uses examples drawn from the pubs database, a sample database included with Microsoft SQL Server and designed to help you test database access code. If you aren't using SQL Server, you won't have this database available. Instead, you can use MSDE, the free data engine included with Visual Studio .NET. MSDE is a scaled-down version of SQL Server that's free to distribute. Its only limitations are that it's restricted to five simultaneous users and that it doesn't provide any graphical tools for creating and managing databases. To install the pubs database in MSDE or SQL Server, you can use the Transact-SQL script included with the samples for this chapter. Alternatively, you can use a different relational database engine, and tweak the examples in this section accordingly.

SQL Basics

When you interact with a data source through ADO.NET, you use the SQL language to retrieve, modify, and update information. In some cases, ADO.NET will hide some of the details for you and even generate the required SQL statements automatically. However, to design an efficient database application with the minimum amount of frustration, you need to understand the basic concepts of SQL.

SQL (Structured Query Language) is a standard data access language used to interact with relational databases. Different databases differ in their support of SQL or add additional features, but the core commands used to select, add, and modify data are common. In a database product like Microsoft 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 will probably only use a few standard types of SQL statements:

- Use a Select statement to retrieve records.
- Use an Update statement to modify records.
- Use an Insert statement to add a new record.
- Use a Delete statement to delete 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.

Experimenting with SQL

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 give you some features for testing queries. If you are using SQL Server, you can try the SQL Query Analyzer, shown in Figure 13-1. This program lets you type in SQL statements, run them, and see the retrieved results, allowing you to test your SQL assumptions and try out new queries.

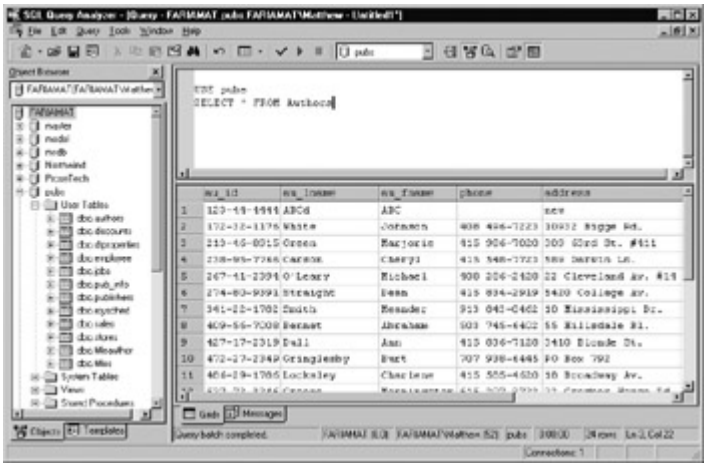


Figure 13-1: The SQL Query Analyzer

More About SQL

You might also want to look at one of the excellent online SQL references available on the Internet. Additionally, if you are working with SQL Server you can use Microsoft's thorough Books Online reference to become a database guru. One topic you might want to investigate is Join queries, which allow you to connect related tables into one set of results.

The SQL Select Statement

To retrieve one or more rows of data, you use a Select statement:

```
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 advanced statements that use sub-grouping, averaging and totaling, and other options (such as setting a maximum number of returned rows). However, though these tasks can be performed with advanced SQL statements, they are often performed manually by your application code or built into a stored procedure in the database.

A Sample Select Statement

A typical (and rather inefficient) Select statement for the pubs database is shown here. 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.

- There is no Where clause. That means that all the records will be retrieved from the database, regardless of whether there are ten or ten million. This is a poor design practice, as it often leads to applications that appear to work fine when they are 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. Often, queries are limited by a date field (for

example, all orders that were placed in the last three months).

- There is no 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 using the tools provided in ADO.NET.

Try These Examples in Query Analyzer

You can try out all the examples in this section without generating an ASP.NET program. Just use the SQL Query Analyzer or a similar query tool. In the Query Analyzer, you can type in the Select statement, and click the green start button. A table of records with the returned information will appear at the bottom of the screen.

Note that you must specify the database you want to use before using any table names. To do this, type the following Use statement first:

```
Use pubs
```

Improving the Select Statement

Here's another example that retrieves a list of author names:

```
SELECT au_lname, au_fname FROM Authors WHERE State='MI' ORDER BY
au_lname ASC
```

- Only two columns are retrieved (au_lname and au_fname). They correspond to the last and first names of the author.

- A Where clause restricts results to those authors who live in the specified state. Note that the Where clause requires apostrophes around the value you want to match (unless it is a numeric value).

- An Order By clause sorts the information alphabetically by the author's last name.

An Alternative Select Statement

Here's one last example:

```
SELECT TOP 100 au_lname, au_fname FROM Authors ORDER BY au_lname,
au_fname ASC
```

- This example uses the Top clause instead of a Where statement. The database rows will be sorted, and the first 100 matching results will be retrieved. The Top clause is useful for user-defined search operations where you want to ensure that the database is not tied up in a long operation retrieving unneeded rows.

- This example uses a more sophisticated Order By expression, which sorts authors with identical last names into a subgroup by their first names.

The Where Clause

In many respects, the Where clause is the most important part of the Select statement. You can combine multiple conditions with the And keyword, and you can 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. The international date format is used to compare date values.

```
SELECT * FROM Sales WHERE ord_date < '2000/01/01' AND ord_date >
'1987/01/01'
```

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 will 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'
```

In the Query Analyzer (see Figure 13-2), this statement will return the number of rows affected, but it will not display the change. To do that, you need to request the row by performing another SQL statement.

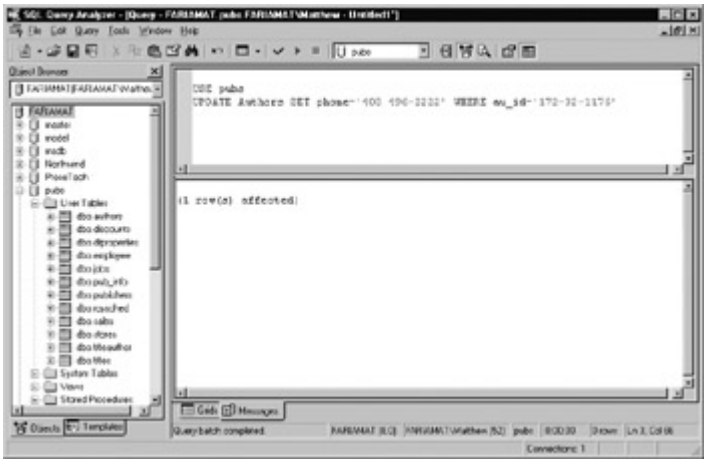


Figure 13-2: An Update statement in the Query Analyzer

As with a Select statement, you can use an Update statement to make several changes at once or to search for a record based on several different 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 modifies the phone number for every author who lives in California:

```
UPDATE Authors SET phone='408 496-2222' WHERE state='CA'
```

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 that 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', 'John', 'Khan', 84152, 0)
```

This example leaves out some information, such as the city and address, in order to provide a simple example. The information shown above is the bare minimum required to create a new record in the Authors table. The new record is featured in Figure 13-3.

Figure 13-3: Inserting a new record

Remember, databases often have special field requirements that may prevent you from adding a record unless you fill in all the values 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 column. This feature, which is supported in most relational database products, assigns a unique value to a specified column when you perform an insert operation. In this case, you shouldn't specify a value for the identity column when inserting the row. Instead, allow the database to choose one automatically.

Auto-Increment Fields Are Indispensable

If you are designing a database, make sure that you add an auto-incrementing identity field to every table. It's the fastest, easiest, and least error-prone way to assign a unique "identification number" to every record. Without an automatically generated identity field, you will 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 for a unique identifier, such as a Social Security number or 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 a SSN number (for example, a foreign national) or account for an SSN number 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 number field). A much better approach is to use a unique identifier, and have the database engine assign an arbitrary unique number to every row automatically.

If you create a table without a unique identification column, you will have trouble when you need to

select that 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 (like apostrophes). You'll also find it extremely awkward to create table relationships.

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: once 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'
```

SQL 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 was successful or executed as expected.

The rest of this chapter shows how you can combine the SQL language with the ADO.NET objects to retrieve and manipulate data in your web applications.

Accessing Data the Easy Way

The "easy way" to access data is to perform all your database operations manually and not worry about maintaining disconnected information. This model is closest to traditional ADO programming, and it allows you to side-step potential concurrency problems, which result when multiple users try to update information at once.

Generally, simple data access is ideal if you only need to read information or if you only need to perform simple update operations, such as adding a record to a log or allowing a user to modify values in a single record (for example, customer information for an e-commerce site). Simple data access is not as useful if you want to provide sophisticated data manipulation features, such as the ability to perform multiple edits on several different records (or several tables).

With simple data access, a disconnected copy of the data is not retained. That means that data selection and data modifications are performed separately. Your program must keep track of the changes that need to be committed to the data source. For example, if a user deletes a record, you need to explicitly specify that record using an SQL Delete statement.

Simple Data Access

To retrieve 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.

Simple Data Updates

To add or update information, follow these steps:

1. Create new Connection and Command objects.

2. Execute the Command (with the appropriate SQL statement).

Importing the Namespaces

Before continuing any further, make sure you import the ADO.NET namespaces, as shown here. Alternatively, you can add these as project-wide imports by modifying your project's properties in Visual Studio .NET.

```
Imports System.Data
Imports System.Data.OleDb  ' Or 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 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 that you can respond if an error does occur.

To create a connection, 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 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.

```
Dim MyConnection As New OleDbConnection()
MyConnection.ConnectionString = "Provider=SQLOLEDB.1;" & _
  "Data Source=localhost;" & _
  "Initial Catalog=Pubs;User ID=sa"
```

The connection string for the SqlConnection object is quite similar, and just leaves out the Provider setting:

```
Dim MyConnection As New SqlConnection()
MyConnection.ConnectionString = "Data Source=localhost;" & _
 "Initial Catalog=Pubs;User ID=sa"
```

The Connection String

The connection string is actually a series of distinct pieces of information, separated by semicolons (;). In the preceding example, the connection string identifies the following pieces of information:

Provider This is the name of the OLE DB provider, which allows communication between ADO.NET and your database. (SQLOLEDB is the OLE DB provider for SQL.) Other providers include MSDAORA (the OLE DB provider for an Oracle database) and

Microsoft.Jet.OLEDB.4.0 (the OLE DB provider for Access).

Data Source This indicates the name of the server where the data source is located. In this case, the server is on the same computer hosting the ASP.NET site, so localhost is sufficient.

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 running an SQL command or by modifying the Database property.

User ID This is used to access the database. The user ID "sa" corresponds to the system administrator account provided with databases such as SQL Server.

Password By default, the sa account doesn't have a password. Typically, in a production-level site, this account would be modified or replaced. To specify a password for a user, just add the Password settings to the connection string (as in "Password=letmein"). Note that as with all connection string settings, you don't need quotation marks, but you do need to separate the setting with a semicolon.

ConnectionTimeout This determines how long your code will wait, in seconds, 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. Use 0 to specify no limit, which is a bad idea. That means that, theoretically, the user's web page could be held up indefinitely while this code attempts to find the server.

SQL Server Integrated Authentication

If the example connection string doesn't allow you to connect to an SQL Server database, it could be because the standard SQL Server accounts have been disabled in favor of integrated Windows authentication (in SQL Server 2000, this is the default).

- With **SQL Server authentication**, SQL Server maintains its own user account information.
- With **integrated authentication**, SQL Server automatically uses the Windows account information for the currently logged-in user. In this case, you would use a connection string with the Integrated Security option:

```
MyConnection.ConnectionString = "Provider=SQLOLEDB.1;" & _
 "Data Source=localhost;" & _
 "Initial Catalog=Pubs;Integrated Security=SSPI"
```

For this to work, the currently logged-on Windows user must have the required authorization to access the SQL database. In the case of an ASP.NET application, the "current user" is set based on IIS and web.config options. For more information, refer to Chapter 24, which discusses security.

Other Connection String Values

There are some other, lesser-used options you can set for a connection string, such as parameters that specify how long you'll wait while trying to make a connection before timing out. For more information, refer to the .NET help files (under SqlConnection or OleDbConnection).

Connection String Tips

Typically, all the database code in your application will use the same connection string. For that

reason, it usually makes the most sense to store a connection string in a globally available variable or class member.

```
Dim ConnectionString As String = "Provider=SQLOLEDB ..."
```

You can also create a Connection object with a preassigned connection string by using a special constructor:

```
Dim MyConnection As New OleDbConnection(ConnectionString)
' MyConnection.ConnectionString is now set to ConnectionString.
```

Making the Connection

Before you can use a connection, you have to explicitly open it:

```
MyConnection.Open()
```

To verify that you have successfully connected to the database, you can try displaying some basic connection information. The following example uses a label control called lblInfo (see Figure 13-4).

Figure 13-4: Testing your connection

Here's the code using a Try/Catch error-handling block:

```
' Define the ADO.NET connection object.
Dim MyConnection As New OleDbConnection(ConnectionString)

Try
   ' Try to open the connection.
   MyConnection.Open()
   lblInfo.Text = "<b>Server Version:</b> "
   lblInfo.Text &= MyConnection.ServerVersion
   lblInfo.Text &= "<br><b>Connection Is:</b> "
   lblInfo.Text &= MyConnection.State.ToString()
Catch err As Exception
   ' 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.
   If (Not Is Nothing MyConnection) Then
      MyConnection.Close()
      lblInfo.Text &= "<br><b>Now Connection Is:</b> "
      lblInfo.Text &=  MyConnection.State.ToString()
   End If
End Try
```

Once 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 to make sure your site runs efficiently.

Closing a connection is just as easy:

```
MyConnection.Close()
```

Defining a Select Command

The Connection object provides a few basic properties that give 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 catch the retrieved records

Command objects represent SQL statements. To use a Command, you define it, specify the SQL statement you want to use, specify an available connection, and execute the command.

You can use one of the earlier SQL statements as shown here:

```
Dim MyCommand As New OleDbCommand()
MyCommand.Connection = MyConnection
MyCommand.CommandText = "SELECT * FROM Authors"
```

Or you can use the constructor as a shortcut:

```
Dim MyCommand As New OleDbCommand("SELECT * FROM Authors", _
  MyConnection)
```

The process is identical for an SqlCommand:

```
Dim MyCommand As New SqlCommand("SELECT * FROM Authors", MyConnection)
```

Using a Command with a DataReader

Once 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 simple data access.

Before you can use a DataReader, make sure you have opened the connection:

```
MyConnection.Open()
```

To create a DataReader, you use the ExecuteReader method of the Command object:

```
' You don't need to use the New keyword because
```

```
' the Command will create the DataReader.
Dim MyReader As OleDbDataReader
MyReader = MyCommand.ExecuteReader()
```

The process is identical for the SqlDataReader:

```
Dim MyReader As SqlDataReader
MyReader = MyCommand.ExecuteReader()
```

These two lines of code define a DataReader and create it using your command. (In order for them to work, you'll need all the code we've created so far in the previous sections to define a connection and a command.)

Once you have the reader, you retrieve a row using the Read method:

```
MyReader.Read()   ' The first row in the result set is now available.
```

You can then access values in that row using the corresponding field name. 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 have 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 have finished reading all the results you need, close the DataReader and Connection:

```
MyDataReader.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 using a drop-down list box, as shown in Figure 13-5. The full record is then retrieved and displayed in a simple label control, as shown in Figure 13-6.

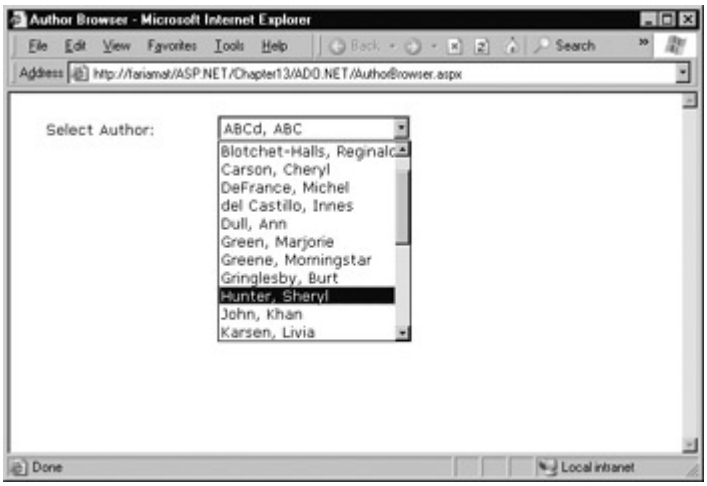


Figure 13-5: Selecting an author

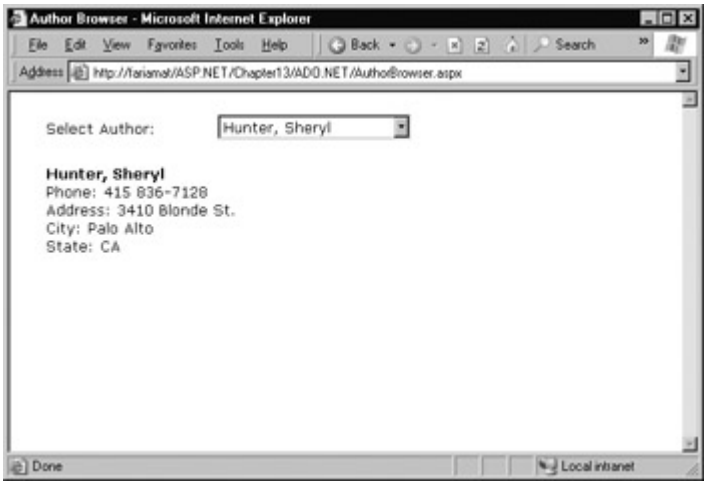


Figure 13-6: Author information

Filling the List Box

To start off, the connection string is defined as a private variable for the page class:

```
Private ConnectionString As String = "Provider=SQLOLEDB.1;" & _
 "Data Source=localhost;Initial Catalog=pubs;Integrated Security=SSPI"
```

The list box is filled after the Page.Load event occurs. Because the list box control is set to persist its viewstate information, this information only needs to be retrieved once, the first time the page is displayed. It will be ignored on all postbacks.

```
Private Sub Page_Load(sender As Object, e As EventArgs) _
  Handles MyBase.Load

    If Me.IsPostBack = False Then
        FillAuthorList()
    End If

End Sub

Private Sub FillAuthorList()

    lstAuthor.Items.Clear()

    ' Define the Select statement.
    ' Three pieces of information are needed: the unique id,
    ' and the first and last name.
    Dim SelectSQL As String
    SelectSQL = "SELECT au_lname, au_fname, au_id FROM Authors"

    ' Define the ADO.NET objects.
    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(SelectSQL, con)
    Dim reader As OleDbDataReader

    ' 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.
        Do While reader.Read()
            Dim NewItem As New ListItem()
            NewItem.Text = reader("au_lname") & ", " & _
              reader("au_fname")
            NewItem.Value = reader("au_id")
            lstAuthor.Items.Add(NewItem)
```

```
        Loop

        reader.Close()

    Catch err As Exception
        lblResults.Text = "Error reading list of names. "
        lblResults.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

End Sub
```

Interesting Facts About this Code

- This example looks more sophisticated than the previous bite-sized snippets in this chapter, but it really doesn't introduce anything new. The standard Connection, Command, and DataAdapter objects are used. The connection is opened inside an error-handling block, so that your page can handle any unexpected errors and provide information. A Finally block makes sure that 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 used to get another row of information. When the reader has read all the available information, this method will return False, the While 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, and 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 list box selection. To accomplish this effect, the AutoPostBack property of the list box is set to True, so its change events are detected automatically.

```
Private Sub lstAuthor_SelectedIndexChanged(sender As Object, _
  e As EventArgs) Handles lstAuthor.SelectedIndexChanged

    ' Create a Select statement that searches for a record
    ' matching the specific author id from the Value property.
    Dim SelectSQL As String
    SelectSQL = "SELECT * FROM Authors "
    SelectSQL &= "WHERE au_id='" & lstAuthor.SelectedItem.Value & "'"

    ' Define the ADO.NET objects.
    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(SelectSQL, con)
    Dim reader As OleDbDataReader

    ' Try to open database and read information.
    Try
        con.Open()
        reader = cmd.ExecuteReader()
        reader.Read()
        lblResults.Text = "<b>" & reader("au_lname")
        lblResults.Text &= ", " & reader("au_fname") & "</b><br>"
        lblResults.Text &= "Phone: " & reader("phone") & "<br>"
```

```
        lblResults.Text &= "Address: " & reader("address") & "<br>"
        lblResults.Text &= "City: " & reader("city") & "<br>"
        lblResults.Text &= "State: " & reader("state") & "<br>"
        reader.Close()
    Catch err As Exception
        lblResults.Text = "Error getting author. "
        lblResults.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

End Sub
```

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 very useful) technique.

- Only one record is read. The code assumes that only one author has the matching au_id, which is reasonable as this field is unique.

What About More Advanced Controls?

This example shows how ADO.NET works to retrieve a simple result set. Of course, ADO.NET also provides handy controls that go beyond this generic level, and let you provide full-featured grids with sorting and editing. These controls are described in Chapter 15. For now, concentrate on understanding the fundamentals about ADO.NET and how it works with data.

Updating Data

Now that you understand how to retrieve data, it's not much more complicated to perform simple delete and update 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 need an SQL Select command. Instead, you can use one of three new SQL commands: Update, Insert, or Delete.

Using Update, Insert, and Delete Commands

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 non-existent table, an exception will occur.)

To demonstrate how to Update, Insert, and Delete simple information, the previous example has been enhanced. Instead of being displayed in a label, the information from each field is added to a separate text box. Two additional buttons allow you to update the record (Update), or delete it (Delete). You can also insert a new record by clicking Create New, entering the information in the text boxes, and then clicking Insert New, as shown in Figure 13-7.

Figure 13-7: A more advanced author manager

The record selection code is identical from an ADO.NET perspective, but it now uses the individual text box controls.

```
Private Sub lstAuthor_SelectedIndexChanged(sender As Object, _
  e As EventArgs) Handles lstAuthor.SelectedIndexChanged

    ' Define ADO.NET objects.
    Dim SelectSQL As String
    SelectSQL = "SELECT * FROM Authors "
    SelectSQL &= "WHERE au_id='" & lstAuthor.SelectedItem.Value & "'"
    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(SelectSQL, con)
    Dim reader As OleDbDataReader

    ' Try to open database and read information.
    Try
        con.Open()
        reader = cmd.ExecuteReader()
        reader.Read()

        ' Fill the controls.
        txtID.Text = reader("au_id")
        txtFirstName.Text = reader("au_fname")
        txtLastName.Text = reader("au_lname")
        txtPhone.Text = reader("phone")
        txtAddress.Text = reader("address")
        txtCity.Text = reader("city")
        txtState.Text = reader("state")
        txtZip.Text = reader("zip")
        chkContract.Checked = CType(reader("contract"), Boolean)
        reader.Close()
        lblStatus.Text = ""
    Catch err As Exception
        lblStatus.Text = "Error getting author. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

End Sub
```

To see the full code, refer to the included example files. If you play with the example at length, you'll notice that it lacks a few niceties that would be needed in a professional web site. 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 using separate pages. In this case, however, the page is useful as a quick way to test some basic data access code.

Updating a Record

When the user clicks the Update button, the information in the text boxes is applied to the database:

```
Private Sub cmdUpdate_Click(sender As Object, _
  e As EventArgs) Handles cmdUpdate.Click

    ' Define ADO.NET objects.
    Dim UpdateSQL As String
    UpdateSQL = "UPDATE Authors SET "
    UpdateSQL &= "au_id='" & txtID.Text & "', "
    UpdateSQL &= "au_fname='" & txtFirstName.Text & "', "
    UpdateSQL &= "au_lname='" & txtLastName.Text & "', "
    UpdateSQL &= "phone='" & txtPhone.Text & "', "
    UpdateSQL &= "address='" & txtAddress.Text & "', "
    UpdateSQL &= "city='" & txtCity.Text & "', "
    UpdateSQL &= "state='" & txtState.Text & "', "
    UpdateSQL &= "zip='" & txtZip.Text & "', "
    UpdateSQL &= "contract='" & Int(chkContract.Checked) & "' "
    UpdateSQL &= "WHERE au_id='" & lstAuthor.SelectedItem.Value & "'"

    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(UpdateSQL, con)

    ' Try to open database and execute the update.
    Try
        con.Open()
        Dim Updated As Integer
        Updated = cmd.ExecuteNonQuery
        lblStatus.Text = Updated.ToString() & " records updated."
    Catch err As Exception
        lblStatus.Text = "Error updating author. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

End Sub
```

The update code is similar to the record selection code. The main differences are:

- No DataReader is used, because no results are returned.
- A dynamically generated Update command is used for the Command object. It selects the corresponding author records, and changes all the fields to correspond to the values entered in the text boxes.
- The ExecuteNonQuery method returns the number of affected records. This information is displayed in a label to confirm to the user that the operation was successful.

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.

```
Private Sub cmdDelete_Click(sender As Object, _
  e As EventArgs) Handles cmdDelete.Click

    ' Define ADO.NET objects.
    Dim DeleteSQL As String
    DeleteSQL = "DELETE FROM Authors "
    DeleteSQL &= "WHERE au_id='" & lstAuthor.SelectedItem.Value & "'"

    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(DeleteSQL, con)

    ' Try to open database and delete the record.
    Dim Deleted As Integer
    Try
        con.Open()
        Deleted = cmd.ExecuteNonQuery()
    Catch err As Exception
        lblStatus.Text = "Error deleting author. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

    ' If the delete succeeded, refresh the author list.
    If Deleted > 0 Then
        FillAuthorList()
    End If

End Sub
```

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 13-8) and does not cause any real problems.

Figure 13-8: 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 it is not linked to any other records, its deletion will be allowed.

Adding a Record

To start adding a new record, click Create New to clear the fields. Technically speaking, this step isn't required, but it simplifies the user's life.

```
Private Sub cmdNew_Click(sender As Object, _
  e As EventArgs) Handles cmdNew.Click

    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."

End Sub
```

The Insert New button performs the actual ADO.NET code to insert the finished record using a dynamically generated Insert statement:

```
Private Sub cmdInsert_Click(sender As Object, _
  e As EventArgs) Handles cmdInsert.Click

    ' Perform user-defined checks.
    ' Alternatively, you could use RequiredFieldValidator controls.
    If txtID.Text = "" Or txtFirstName.Text = "" Or _
      txtLastName.Text = "" Then

        lblStatus.Text = "Records require an ID, first name,"
        lblStatus.Text &= "and last name."
```

```
        Exit Sub
    End If

    ' Define ADO.NET objects.
    Dim strInsert As String
    strInsert = "INSERT INTO Authors ("
    strInsert &= "au_id, au_fname, au_lname, "
    strInsert &= "phone, address, city, state, zip, contract) "
    strInsert &= "VALUES ('"
    strInsert &= txtID.Text & "', '"
    strInsert &= txtFirstName.Text & "', '"
    strInsert &= txtLastName.Text & "', '"
    strInsert &= txtPhone.Text & "', '"
    strInsert &= txtAddress.Text & "', '"
    strInsert &= txtCity.Text & "', '"
    strInsert &= txtState.Text & "', '"
    strInsert &= txtZip.Text & "', '"
    strInsert &= Int(chkContract.Checked) & "')"

    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(strInsert, con)

    ' Try to open database and execute the update.
    Dim Added As Integer
    Try
        con.Open()
        Added = cmd.ExecuteNonQuery
        lblStatus.Text = Added.ToString() & " records inserted."
    Catch err As Exception
        lblStatus.Text = "Error inserting record. "
        lblStatus.Text &= err.Message
    Finally
        If (Not con Is Nothing) Then
            con.Close()
        End If
    End Try

    ' If the insert succeeded, refresh the author list.
    If Added > 0 Then
        FillAuthorList()
    End If

End Sub
```

If the insert fails, the problem will be reported to the user in a rather unfriendly way (see Figure 13-9). This is typically a result of not specifying valid values. In a more polished application, you would use validators (as discussed in Chapter 9) and provide more useful error messages.

Figure 13-9: A failed insertion

If the insert operation is successful, the page is updated with the new author list.

Accessing Disconnected Data

With disconnected data manipulation, your coding is a little different. In many cases, you have less SQL code to write, and you modify records through the DataSet object. However, you need to carefully watch for problems that can occur when you send changes to the data source. In our simple one-page scenario, disconnected data access won't present much of a problem. If, however, you use disconnected data access to make a number of changes and commit them all at once, you are more likely to run into trouble.

With disconnected data access, a copy of the data is retained, briefly, in memory while your code is running. Changes are tracked automatically using the built-in features of the DataSet object. You fill the DataSet in much the same way that you connect a DataReader. However, while the DataReader held a live connection, information in the DataSet is always disconnected.

Selecting Disconnected Data

The following example shows how you could rewrite the FillAuthorList() subroutine to use a DataSet instead of a DataReader. The changes are highlighted in bold.

```
Private Sub FillAuthorList()

    lstAuthor.Items.Clear()

    ' Define ADO.NET objects.
    Dim SelectSQL As String
    SelectSQL = "SELECT au_lname, au_fname, au_id FROM Authors"
    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(SelectSQL, con)
    Dim adapter As New OleDbDataAdapter(cmd)
    Dim Pubs As New DataSet()

    ' Try to open database and read information.
    Try
        con.Open()
        ' All the information in transferred with one command.
```

UNIT - 5
DATA BINDING

Chapter 14: Data Binding

Overview

In Chapter 13, you learned how to use ADO.NET to retrieve information from a database, work with it in an ASP.NET application, and then apply your changes back to the original data source. 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, it still requires a lot of repetitive code to move through the data, format columns, and display it in the correct order. Repetitive code may be easy, but it's also error-prone, difficult to enhance, and unpleasant to look at. 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.

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 is superficially similar to data binding in the world of desktop or client/server applications, but fundamentally different. In those environments, data binding refers to the creation of 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. It's for these reasons that data binding has acquired a bad reputation.

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 objects are thrown away and the page is sent to the client. If the user modifies the data in a databound control, your program can update the database, but nothing happens automatically.

ASP.NET data binding is much more flexible than traditional data binding. Many of the most powerful data binding controls, such as the Repeater, DataList, and DataGrid, allow you to configure formatting options and even add repeating controls and buttons for each record. This is all set up through special templates, which are a new addition to ASP.NET. Templates are examined in detail in Chapter 15.

Types of ASP.NET Data Binding

There are actually two types of ASP.NET data binding.

Single-Value or "Simple" Data Binding

This type of data binding is used 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. In fact, because ADO.NET usually works with entire lists or tables of information, single-value data binding is rarely of much use. Instead, single-value data binding allows you to take a variable, property, or expression and

insert it dynamically into a page.

Repeated-Value or List Binding

This type of data binding is much more useful for handling complex tasks, such as displaying an entire table or all the values from a single field in a table. Unlike single-value data binding, this type of data binding requires a special control that supports it. Typically, this will be a list control like CheckBoxList or ListBox, but it can also be a much more sophisticated control like the DataGrid. You will know that a control supports repeated-value data binding if it provides a DataSource property. As with single-value binding, repeated-value binding does not 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 are using single-value or repeated-value binding. In single-value binding, a data binding expression is inserted right into the display portion of the .aspx file (not the code portion or code-behind file). In repeated-value binding, data binding is configured by setting the appropriate control properties (for example, in the Load event for the page). You'll see specific examples of both these techniques later in this chapter.

Once 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 for the specific list control you are using. Alternatively, you can bind the whole page at once by calling the DataBind method for the current page. Once you call this method, all the data binding expressions in the page are evaluated and replaced with the specified value.

Typically, you use the DataBind method in the Page.Load event. 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 very general description of the whole process. To really understand what's happening, you need to work with some specific examples.

Single-Value Data Binding

Single-value data binding is really just a different approach to dynamic text. To use it, you add a special data binding expression 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 special tag, you will receive an error. The only thing you can add is valid data binding expressions. For example, if you have a public variable on your page named Country, you could write:

```
<%# 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:

```
<%# Request.Browser.Browser %>
```

This would substitute a string with the current browser name (for example, IE). In fact, you can

even call a built-in function, a public 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. The following data binding expressions are all valid:

```
<%# GetUserName(ID) %>
<%# 1 + (2 * 20) %>
<%# "John " & "Smith" %>
```

Remember, these data binding expressions are placed in the HTML code for your .aspx file. That means that if you're relying on Visual Studio .NET to manage your HTML code automatically, you may have to venture into slightly unfamiliar territory. To examine how you can add a data binding expression, and see why you might want to, it helps to review a simple example.

A Simple Data Binding Example

This section considers 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 special variable defined in your Page class, which is called TransactionsCount:

```
Public Class DataBindingPage
  Inherits Page

    Public TransactionCount As Integer

    ' (Additional code omitted.)

End Class
```

Note that this variable must be designated as Public, not Private. Otherwise, ASP.NET will not be able to access it when it is evaluating the data binding expression.

Now, assume that this value is set in the Page.Load event handler using some database lookup code. For testing purposes, our example skips this step and hard-codes a value:

```
Private Sub Page_Load(sender As Object, e As EventArgs) _
  Handles MyBase.Load

    ' 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.
    Me.DataBind()

End Sub
```

Two things actually happen in this event handler: the TransactionCount variable is set to 10, and all the data binding expressions on the page are bound. Currently, there aren't any data binding expressions, so this method has no effect. Notice that this example uses the Me keyword to refer to the current page. You could just write DataBind() without the Me keyword, as the default object is the current Page object. However, using the Me keyword makes it a bit clearer what control is being used.

To actually 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 presentation code in the .aspx file. If you are using Notepad to create your ASP.NET pages, you won't have any trouble with this approach. If, on the other hand, you're using Visual Studio .NET to create your controls, you should probably add a label control, and then configure the data binding expression in the HTML view by clicking the HTML button (see Figure 14-1).

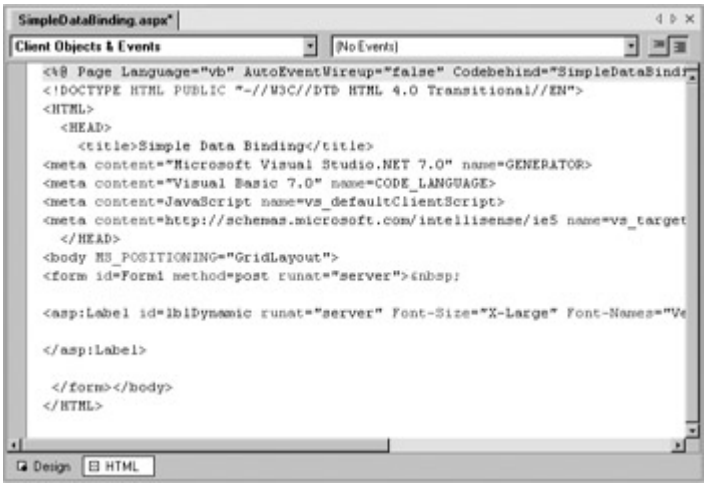


Figure 14-1: HTML view

You could also type this data binding expression in the Properties window, but Visual Studio .NET does not always refresh this information correctly. In this case, it's better to work on a slightly lower level.

To add your expression, find the tag for the label control. Modify the Text property as shown in the following code. Note that some of the label attributes (such as its size and position) have been left out to help make it clearer. You should not, however, delete this information in your code!

```
<asp:Label id=lblDynamic runat="server" Font-Size="X-Large"
Font-Names="Verdana">
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 intTransactionsToday variable, and the second uses the built-in Request object to find out some information about the user's browser. When you run this page, the output looks like Figure 14-2.

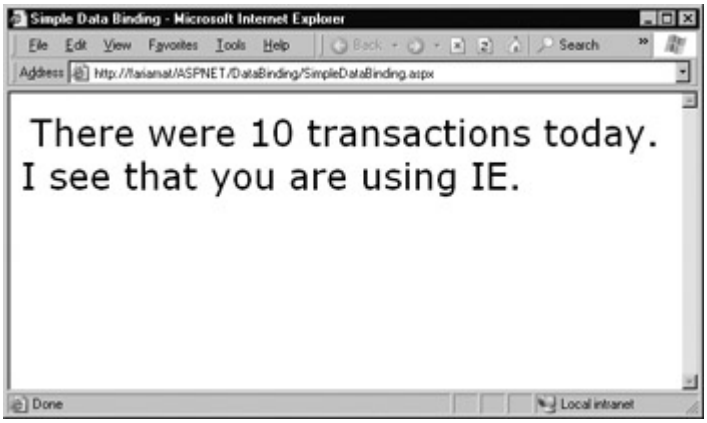


Figure 14-2: The result of data binding

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 called the DataBind method, that information would be popped into the page in the locations you have designated.

If, however, you forgot to call the DataBind method, the expressions would be ignored, and the user would see a somewhat confusing window that looks like Figure 14-3.

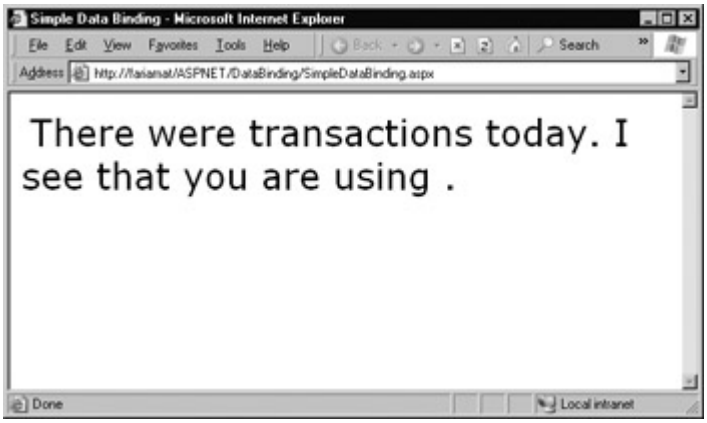


Figure 14-3: The non-databound page

You also need to be careful with your databound control in the design environment. If you modify its properties in the Properties window of Visual Studio .NET, the editor can end up thoughtlessly obliterating your data binding expressions.

Order Is Important

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 into 0. Remember, data binding is a one-way street. That means there won't be any visible effect if you change the TransactionCount variable after you've used the DataBind method. Unless you call the DataBind method again, the displayed value won't be updated.

Simple Data Binding with Properties

In the example so far, data binding expressions are used 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.

For example, consider a page that defines a URL variable and uses it to refer to a picture in the application directory:

```
Public Class DataBindingPage
  Inherits Page

    Public URL As String

    Private Sub Page_Load(sender As Object, e As EventArgs) _
      Handles MyBase.Load
        URL = Server.MapPath("picture.jpg")
        Me.DataBind()
    End Sub

End Class
```

This URL can now be used to create a label as the previous example did:

```
<asp:Label id=lblDynamic runat="server"><%# URL %></asp:Label>
```

You can also use it as a caption for a checkbox:

```
<asp:CheckBox id="chkDynamic" Text="<%# URL %>" runat="server" />
```

Or as a target for a hyperlink control:

```
<asp:Hyperlink id="lnkDynamic" Text="Click here!"
    NavigateUrl="<%# URL %>" runat="server" />
```

Or even as a picture:

```
<asp:Image id="imgDynamic" Src="<%# URL %>" runat="server" />
```

The only trick is that you need to be able to edit these control tags manually. A final combined page that uses all these elements would look something like Figure 14-4.

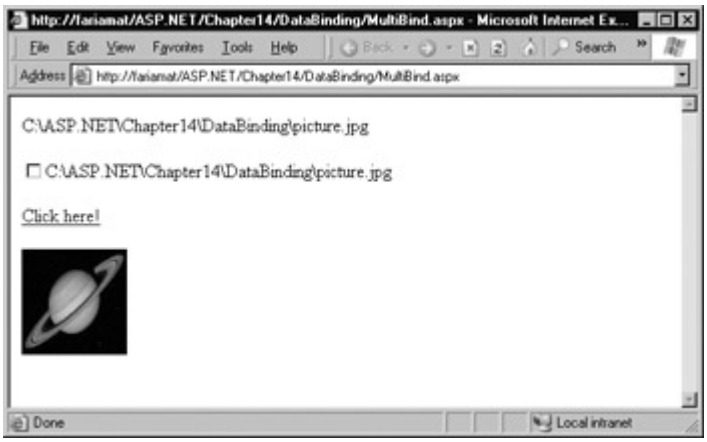


Figure 14-4: Multiple ways to bind the same data

To examine it in more detail, try out 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 finally 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, over-enthusiastic 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.

Code fragmentation When data binding expressions are used, it may not be obvious to other developers where the functionality resides for different operations. This is particularly a problem if you blend both approaches (modifying the same control using a data binding expression, and then 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.

No design-time error checking When you type a data binding expression, you need to remember the corresponding variable, property, or method name and enter it exactly. If you don't, you won't realize your mistake until you try to run the page and you receive an error (see Figure 14-5). This is a significant drawback for those who are

used to Visual Studio .NET's automatic error checking. You also can't take advantage of Visual Studio .NET's IntelliSense features to automatically complete code statements.

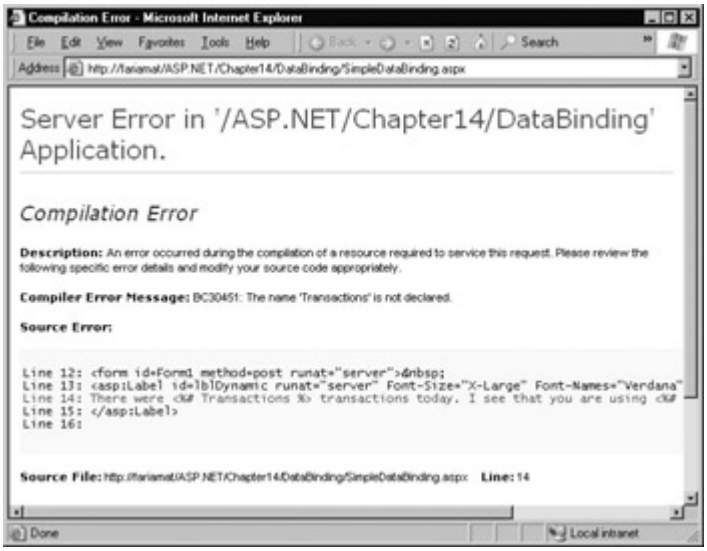


Figure 14-5: A common error

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. Often, single-value data binding suits developers who prefer to write code directly in .aspx files without Visual Studio .NET. These developers manually write all the control tags they need, and can thus use single-value data binding to create a shorter, more concise file without adding extra code.

Using Code Instead of Simple Data Binding

If you decide not to use single-value data binding, you can accomplish the same thing using code. For example, you could use the following event handler to display the same output that our first label example created.

```
Private Sub Page_Load(sender As Object, e As EventArgs) _
  Handles MyBase.Load

    TransactionCount = 10
    lblDynamic.Text = "There were " & TransactionCount.ToString()
    lblDynamic.Text &= "transactions today. "
    lblDynamic.Text &= "I see that you are using " & _
      Request.Browser.Browser

End Sub
```

This code dynamically fills in the label without using data binding. The trade-off is more code. Instead of importing ASP.NET code into the .aspx file, you end up doing the reverse: importing user interface (the specific text) into your code file!

Stick to One Approach

As much as possible, stick to one approach for using dynamic text. If you decide to use data binding, try to reduce the number of times you modify a control's text in code. If you do both, you may end up confusing others, or just canceling your own changes! For example, if you call the DataBind method on a control that uses a data expression after changing its values in code, your data binding expression will not be used.

Repeated-Value Data Binding

While using simple data binding is an optional choice, repeated-value binding is so useful that almost every ASP.NET application will want to make use of it somewhere. Repeated-value data binding uses one of the special list controls included with ASP.NET. You link one of these controls to a data list source (such as a field in a data table), and the control automatically creates a full list using all the corresponding values. This saves you from having to write 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 is 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 whole collection, many of which you've probably already used in other applications or examples:

The ListBox, DropDownList, CheckBoxList, and RadioButtonList web controls These controls provide a list for a single-column of information.

The HtmlSelect server-side HTML control This control represents the HTML <select> element, and works essentially the same as the ListBox web control. Generally, you'll only use this control for backward compatibility or when upgrading an existing ASP page.

The DataList, DataGrid, and Repeater controls These controls allow you to provide repeating lists or grids that can display more than one column (or field) of information at a time. For example, if you were binding to a hashtable (a special type of collection), you could display both the key and value of each item. If you were binding to a full-fledged table in a DataSet, you could display multiple fields in any combination. 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 will usually just set properties. Of course, you can set properties in many ways, using code in a code-behind file, or by modifying the control tag in the .aspx file, possibly with the help of Visual Studio .NET's Properties window. The approach you take does not matter. The important detail is that you don't use any <%# expression %> data binding expressions.

For more sophisticated and flexible controls, such as the Repeater and DataList, you need to set some properties *and* enter data binding expressions in the .aspx file, generally using a special template format. Understanding the use of templates is the key to mastering data binding, and we'll explore it in detail in Chapter 15.

To continue any further with data binding, it helps to divide the subject into a few basic categories. We'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 only need to follow three steps:

1. **Create and fill some kind of data object.** There are numerous options, including an Array, ArrayList, Collection, Hashtable, DataTable, and DataSet. Essentially, you can use any type of collection that supports the IEnumerable interface, although there are specific advantages

and disadvantages that you will discover in each class.

2. **Link the object to the appropriate control.** To do this, you only need to set a couple of properties, including DataSource. If you are binding to a full DataSet, you will also need to set the DataMember property to identify the appropriate table that 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 are using the ListBox, DropDownList, CheckBoxList, RadioButtonList, or even HtmlSelect control. All these controls provide the exact 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 out this type of data binding, add a ListBox control to a new web page. Use the Page.Load event handler to create a collection to use as a data source:

```
Dim colFruit As New Collection()
colFruit.Add("Kiwi")
colFruit.Add("Pear")
colFruit.Add("Mango")
colFruit.Add("Blueberry")
colFruit.Add("Apricot")
colFruit.Add("Banana")
colFruit.Add("Peach")
colFruit.Add("Plum")
```

Now, you can link this collection to the list box control.

```
lstItems.DataSource = colFruit
```

Because a collection is a straightforward, unstructured type of object, this is all the information you need to set. If you were using a DataTable (which has more than one field) or a DataSet (which has more than one DataTable), you would have to specify additional information, as you will see in later examples.

To activate the binding, use the DataBind method:

```
Me.DataBind()
' Could also use lstItems.DataBind to bind just the list box.
```

The resulting web page looks like Figure 14-6.

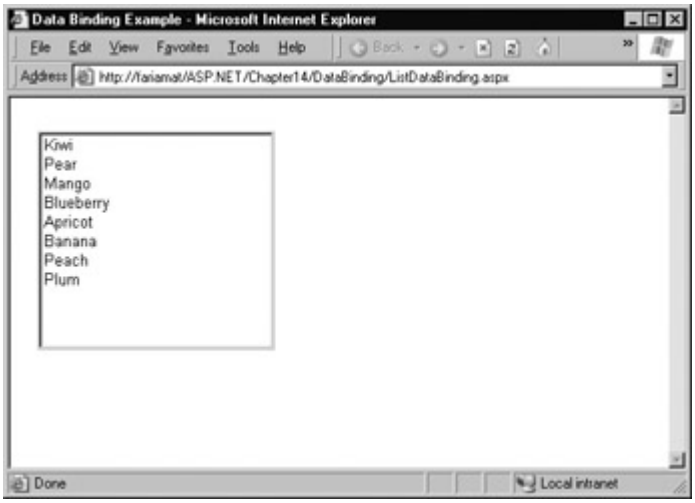


Figure 14-6: A databound list

This technique can save quite a few lines of code. In this example, there isn't a lot of savings because the collection is created just before it is displayed. In a more realistic application, however, you might be using a function that returns a ready-made collection to you.

```
Dim colFruit As Collection    ' No New keyword is needed this time.
colFruit = GetFruitsInSeason("Summer")
```

In this case, it is extremely simple to add the extra two lines needed to bind and display the collection in the window.

```
lstItems.DataSource = colFruit
Me.DataBind()
```

Or even change it to this:

```
lstItems.DataSource = GetFruitsInSeason("Summer")
Me.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 adds up rapidly, especially when you start to combine data binding with multiple controls, advanced objects like DataSets, or advanced controls that apply formatting through templates.

Multiple Binding

You can bind the same data list object to multiple different controls. Consider the following example, which compares all the different types of list controls at your disposal by loading them with the same information:

```
Private Sub Page_Load(sender As Object, e As EventArgs) _
  Handles MyBase.Load

    ' Create and fill the collection.
    Dim colFruit As New Collection()
    colFruit.Add("Kiwi")
    colFruit.Add("Pear")
    colFruit.Add("Mango")
    colFruit.Add("Blueberry")
    colFruit.Add("Apricot")
    colFruit.Add("Banana")
    colFruit.Add("Peach")
    colFruit.Add("Plum")

    ' Define the binding for the list controls.
```

```
    MyListBox.DataSource = colFruit
    MyDropDownListBox.DataSource = colFruit
    MyHTMLSelect.DataSource = colFruit
    MyCheckBoxList.DataSource = colFruit
    MyRadioButtonList.DataSource = colFruit

    ' Activate the binding.
    Me.DataBind()

End Sub
```

The corresponding compiled page looks like Figure 14-7.

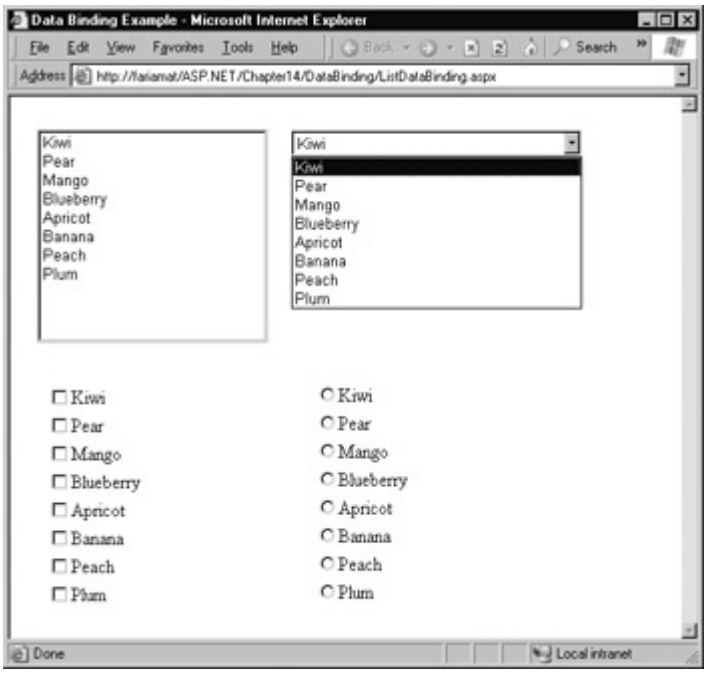


Figure 14-7: Multiple-bound lists

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 only work with 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 would automatically refresh so that they too have Pear selected (or the corresponding information from the same row). This is not how ASP.NET uses data binding.

Viewstate

Remember, the original collection is destroyed as soon as the page is completely processed into HTML and sent to the user. However, the information will remain in the controls if you have set their EnableViewstate properties to True. That means that you don't need to recreate and rebind the control every time the Page.Load event occurs, and you should check the IsPostBack property first.

Of course, in many cases, especially when working with databases, you will want to rebind on every pass. For example, if you presented information corresponding to values in a database table, you might want to let the user make changes or specify a record to be deleted. As soon as the page is posted back, you would then want to execute an SQL command and rebind the control to show the new data (thereby confirming to the user that the data source was updated with the change). In this case, you should disable viewstate, because you will be rebinding the data with every postback.

Chapter 15 will examine this situation a little more closely. The important concept to realize now is

that you need to be consciously evaluating state options. If you don't need viewstate for a databound list control, you should disable it, as it can slow down response times if a large amount of data is displayed on the screen. This is particularly true for our example with multiple binding, as each control will have its own viewstate and its own separate copy of the identical data.

Data Binding with a Hashtable

Collections are not the only kind of data source you can use with list data binding. Other popular choices include various types of arrays and specialized collections. The concepts are virtually identical, so this chapter won't spend much time looking at different possibilities. Instead, we'll consider one more example with a hashtable before continuing on to DataSets and DataReaders, which allow information to be displayed from a database.

A hashtable is a special kind of collection where every item has a corresponding key. It's also called a dictionary, because 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 like Session, Application, and Cache work.

You might already know that the standard collection used in the previous example allows you to specify a key for every inserted item. However, it doesn't require one. This is the main difference between a generic collection and the hashtable—hashtables always need keys, which makes them more efficient for retrieval and sorting. Generic collections, on the other hand, are like large black sacks. Generally, you need to go through every item in a generic collection to find what you need, which makes them ideal for cases where you always need to display or work with all the items at the same time.

You create a hashtable in much the same way that you create a collection. The only difference is that you need to supply a unique key for every item. The next example uses the lazy practice of assigning a sequential number for each key:

```
Private Sub Page_Load(sender As Object, e As EventArgs) _
  Handles MyBase.Load

    If Me.IsPostBack = False Then
        Dim colFruit As New Hashtable()
        colFruit.Add(1, "Kiwi")
        colFruit.Add(2, "Pear")
        colFruit.Add(3, "Mango")
        colFruit.Add(4, "Blueberry")
        colFruit.Add(5, "Apricot")
        colFruit.Add(6, "Banana")
        colFruit.Add(7, "Peach")
        colFruit.Add(8, "Plum")

        ' Define the binding for the list controls.
        MyListBox.DataSource = colFruit

        ' Activate the binding.
        Me.DataBind()
    End If

End Sub
```

However, when you try to run this code, you'll receive the unhelpful page shown in Figure 14-8.

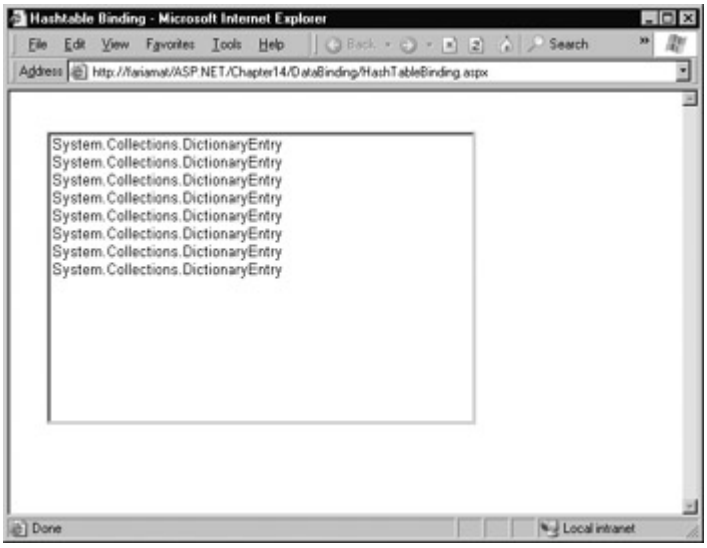


Figure 14-8: Unhelpful data binding

What happened? ASP.NET doesn't know how to handle hashtables without additional information. Instead, it defaults to using the ToString method to get the class name for each item in the collection, which is probably not what you had in mind.

To solve this problem, you have to specify the property that you are interested in. Add this line before the Me.DataBind() statement:

```
' Specify that you will use the Value property of each item.
MyListBox.DataTextField = "Value"
```

The page will now appear as expected, with all the fruit names. Notice that you need to enclose the property name in quotation marks. ASP.NET uses reflection to inspect your object and find the property that matches this string 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 information.

Using the DataValueField Property

Along with the DataTextField property, all list controls that support data binding also provide a DataValueField, 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:

```
MyListBox.DataTextField = "Value"
MyListBox.DataValueField = "Key"
```

The control will appear exactly the same, with a list of all the fruit names in the hashtable. If you look at the HTML source of the page, though, you will see that value attributes have been set with the corresponding numeric key for each item:

```
<select name="MyListBox" id="MyListBox" >
   <option value="8">Plum</option>
   <option value="7">Peach</option>
   <option value="6">Banana</option>
   <option value="5">Apricot</option>
   <option value="4">Blueberry</option>
   <option value="3">Mango</option>
   <option value="2">Pear</option>
```

```
    <option value="1">Kiwi</option>
</select>
```

You can retrieve this value later on using the SelectedItem class to get additional information. For example, you could enable AutoPostBack for the list control, and add the following code:

```
Private Sub MyListBox_SelectedIndexChanged(sender As Object, _
 e As EventArgs) Handles MyListBox.SelectedIndexChanged
    lblMessage.Text = "You picked: " & MyListBox.SelectedItem.Text
    lblMessage.Text &= " which has the key: " & _
      MyListBox.SelectedItem.Value
End Sub
```

The result is demonstrated in Figure 14-9. 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.

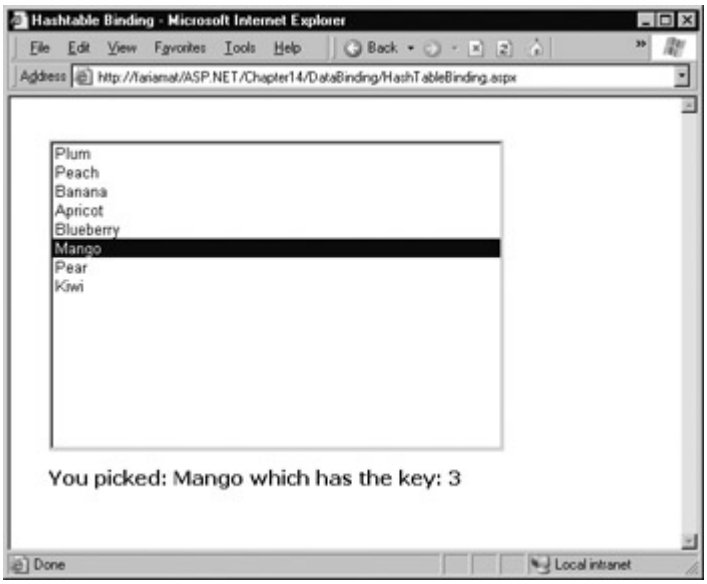


Figure 14-9: Binding to the text and value properties

Note that in order for this to work, you must *not* be regenerating the list with every postback. If you are, the selected item information will be lost, and an error will occur. The preceding example uses the Page.IsPostBack property to determine whether to build the list.

Data Binding with Databases

So far the examples in this chapter have dealt with data binding that doesn't involve databases or any part of ADO.NET. While this is an easy way to familiarize yourself with the concepts, and a useful approach in its own right, the greatest advantage from data binding is achieved when you use it in conjunction with a database.

Remember, in order to work with databases you should import the related namespaces:

```
Imports System.Data
Imports System.Data.OleDb     ' Or System.Data.SqlDB
```

The data binding process still takes place in the same three steps with a database. 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, so a DataSet is a more common choice. In the following example, the DataSet is filled by hand, but it could just as easily be filled with a DataAdapter object.

```
' Define a DataSet with a single DataTable.
Dim dsInternal As 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.
Dim rowNew As 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, a table from the DataSet is bound to the appropriate control. In this case, you need to specify the appropriate field using 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 this case, you have to select a table by setting the control's DataMember property. This is an equivalent approach, but the code looks slightly different:

```
' Define the binding.
lstUser.DataSource = dsInternal
lstUser.DataMember = "Users"
lstUser.DataTextField = "Name"
```

As always, the last step is to activate the binding:

```
Me.DataBind()
' You could also use lstItems.DataBind to bind just the list box.
```

The final result is a list with the information from the specified database field, as shown in Figure 14-10. 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.

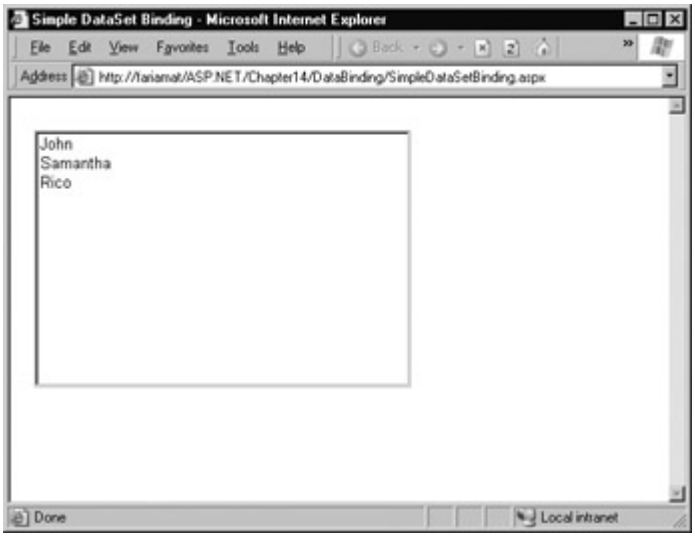


Figure 14-10: DataSet binding

Can You Include More than One Field in a List Control?

The simple list controls require you to bind their Text or Value property to a single data field in the data source object. However, much more flexibility is provided by the more advanced data binding controls examined in Chapter 15. They allow fields to be combined in any way using templates.

Creating a Record Editor

The next example is more practical. It's a good example of how data binding might be used in a full ASP.NET application. This example allows the user to select a record and update one piece of information by using databound list controls. This example uses the Products table from the Northwind database included with SQL Server. A Private variable in the Page class is used to define a connection string that every part of the code can access:

```
Private ConnectionString As String = "Provider=SQLOLEDB.1;" & _
 "DataSource=localhost;Initial Catalog=Northwind;" & _
 "Integrated Security=SSPI"
```

The first step is to create a drop-down list that allows the user to choose a product for editing. The Page.Load event handler takes care of this task, retrieving the data, binding it to the drop-down list control, and then activating the binding:

```
Private Sub Page_Load(sender As Object, e As EventArgs) _
  Handles MyBase.Load

    If Me.IsPostBack = False Then
        ' Define the ADO.NET objects for selecting Products.
        Dim SelectCommand As String
        SelectCommand = "SELECT ProductName, ProductID FROM Products"
        Dim con As New OleDbConnection(ConnectionString)
        Dim com As New OleDbCommand(SelectCommand, con)

        ' Open the connection.
        con.Open()

        ' Define the binding.
        lstProduct.DataSource = com.ExecuteReader()
        lstProduct.DataTextField = "ProductName"
        lstProduct.DataValueField = "ProductID"

        ' Activate the binding.
```

```
        lstProduct.DataBind()

        con.Close()

        ' Make sure nothing is currently selected.
        lstProduct.SelectedIndex = -1

    End If

End Sub
```

The actual database code is very similar to what we covered in Chapter 13. The example uses a Select statement, but carefully limits the returned information to just the ProductName field, which is the only piece of information it will use. The resulting window shows a list of all the products defined in the database, as shown in Figure 14-11.

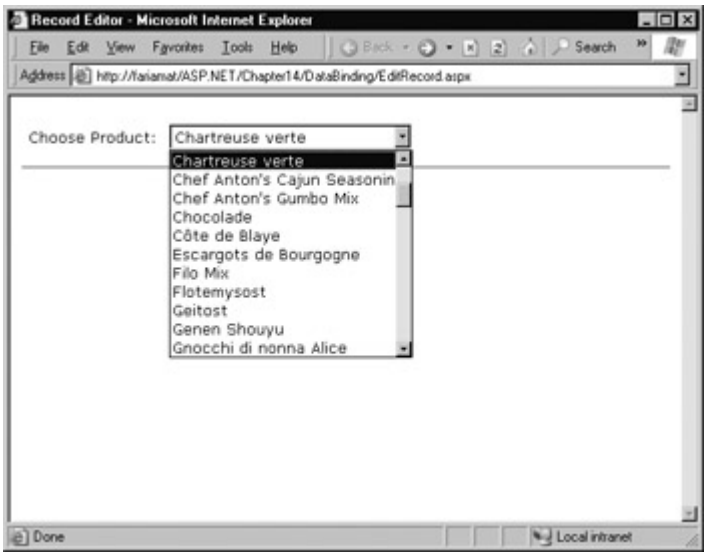


Figure 14-11: Product choices

The drop-down list enables AutoPostBack, so as soon as the user makes a selection, a lstProduct.SelectedItemChanged event fires. At this point, our code performs several tasks:

- It reads the corresponding record from the Products table and displays additional information about it in a label control. In this case, a special Join query is used to link information from the Products and Categories table. 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.

- It highlights the row in the category list box 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 is really just an application of what you have learned over the last two chapters. The full listing is shown here. Remember, all this code is placed inside the event handler for the lstProdcut.SelectedIndexChanged.

```
' Create a command for selecting the matching product record.
Dim ProductCommand As String
ProductCommand = "SELECT ProductName, QuantityPerUnit, " & _
 "CategoryName FROM Products INNER JOIN Categories ON " & _
 "Categories.CategoryID=Products.CategoryID " & _
 "WHERE ProductID='" & lstProduct.SelectedItem.Value & "'"
```

```
' Create the Connection and Command objects.
Dim con As New OleDbConnection(ConnectionString)
Dim comProducts As New OleDbCommand(ProductCommand, con)

' Retrieve the information for the selected product.
con.Open()
Dim reader As OleDbDataReader = comProducts.ExecuteReader()
reader.Read()

' Update the display.
lblRecordInfo.Text = "<b>Product:</b> "
lblRecordInfo.Text &= reader("ProductName") & "<br>"
lblRecordInfo.Text &= "<b>Quantity:</b> "
lblRecordInfo.Text = reader("QuantityPerUnit") & "<br>"
lblRecordInfo.Text &= "<b>Category:</b> " & reader("CategoryName")

' Store the corresponding CategoryName for future reference.
Dim MatchCategory As String = reader("CategoryName")

' Close the reader.
reader.Close()

' Create a new Command for selecting categories.
Dim CategoryCommand As String = "SELECT CategoryName, & _
 "CategoryID FROM Categories"
Dim comCategories As New OleDbCommand(CategoryCommand, con)

' Retrieve the category information, and bind it.
lstCategory.DataSource = comCategories.ExecuteReader()
lstCategory.DataTextField = "CategoryName"
lstCategory.DataValueField = "CategoryID"
lstCategory.DataBind()
con.Close()

' Highlight the matching category in the list.
lstCategory.Items.FindByText(MatchCategory).Selected = True

lstCategory.Visible = True
cmdUpdate.Visible = True
```

This code could be improved 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 ProductName and returns a single DataRow with the associated product information. Another improvement would be to use a stored procedure to retrieve this information, rather than a full-fledged DataReader.

The end result is a window that updates itself dynamically whenever a new product is selected, as shown in Figure 14-12.

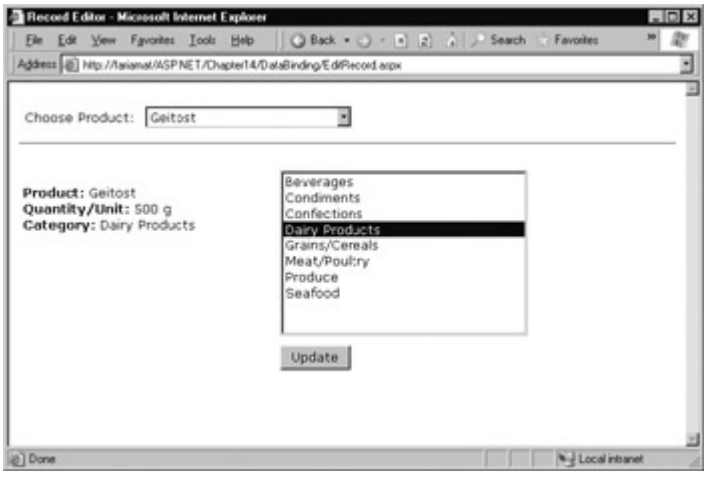


Figure 14-12: Product information

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:

```
Private Sub cmdUpdate_Click(sender As Object, _
 e As EventArgs) Handles cmdUpdate.Click

    ' Define the Command.
    Dim UpdateCommand As String = "UPDATE Products " & _
     "SET CategoryID='" & stCategory.SelectedItem.Value & "' " & _
     "WHERE ProductName='" & lstProduct.SelectedItem.Text & "'"

    Dim con As New OleDbConnection(ConnectionString)
    Dim com As New OleDbCommand(UpdateCommand, con)

    ' Perform the update.
    con.Open()
    com.ExecuteNonQuery()
    con.Close()

End Sub
```

This example could easily be extended so that all the properties in a product record are easily editable, but before you try that, you might want to experiment with template-based data binding, which is introduced in Chapter 15. Using templates, you can create sophisticated lists and grids that provide automatic features for selecting, editing, and deleting records.