

# ASP.NET: Tips, Tutorials, and Code

---

Scott Mitchell, Bill Anders, Rob Howard,  
Doug Seven, Stephen Walther,  
Christop Wille, and Don Wolthuis



**DRAFT**

**SAMS**

*201 West 103rd St., Indianapolis, Indiana, 46290 USA*

0-672-32143-2  
Spring 2001

**DRAFT**

# Common ASP.NET Code Techniques

CHAPTER

2

## IN THIS CHAPTER

- Using Collections 4
- Working with the File System 29
- Using Regular Expressions 45
- Generating Images Dynamically 51
- Sending E-mail from an ASP.NET Page 60
- Network Access Via an ASP.NET Page 64
- Uploading Files from the Browser to the Web Server Via an ASP.NET Page 71
- Using ProcessInfo: Retrieving Information About a Process 79
- Accessing the Windows Event Log 84
- Working with Server Performance Counters 93
- Encrypting and Decrypting Information 101

## Using Collections

Most modern programming languages provide support for some type of object that can hold a variable number of elements. These objects are referred to as collections, and they can have elements added and removed with ease without having to worry about proper memory allocation. If you've programmed with classic ASP before, you're probably familiar with the `Scripting.Dictionary` object, a collection object that references each element with a textual key. A collection that stores objects in this fashion is known as a *hash* table.

There are many types of collections in addition to the hash table. Each type of collection is similar in purpose: it serves as a means to store a varying number of elements, providing an easy way, at a minimum, to add and remove elements. Each different type of collection is unique in its method of storing, retrieving, and referencing its various elements.

The .NET Framework provides a number of collection types for the developer to use. In fact, an entire namespace, `System.Collections`, is dedicated to collection types and helper classes. Each of these collection types can store elements of type `Object`. Because in .NET all primitive data types—string, integers, date/times, arrays, and so on—are derived from the `Object` class, these collections can literally store anything! For example, you could use a single collection to store a couple of integers, an instance of a classic COM component, a string, a date/time, and two instances of a custom-written .NET component. Most of the examples in this section use collections to house primitive data types (strings, integers, doubles). However, Listing 2.1 illustrates a collection of collections—that is, a collection type that stores entire collections as each of its elements!

Throughout this section we'll examine five collections the .NET Framework offers developers: the `ArrayList`, the `Hashtable`, the `SortedList`, the `Queue`, and the `Stack`. As you study each of these collections, realize that they all have many similarities. For example, each type of collection can be iterated through element-by-element using a `For Each . . . Next` loop in VB (or a `foreach` loop in C#). Each collection type has a number of similarly named functions that perform the same tasks. For example, each collection type has a `Clear` method that removes all elements from the collection, and a `Count` property that returns the number of elements in the collection. In fact, the last subsection “Similarities Among the Collection Types” examines the common traits found among the collection types.

### Working with the `ArrayList` Class

The first type of collection we'll look at is the `ArrayList`. With an `ArrayList`, each item is stored in sequential order and is indexed numerically. In our following examples, keep in mind that the developer need not worry himself with memory allocation. With the standard array, the

developer cannot easily add and remove elements without concerning himself with the size and makeup of the array. With all the collections we'll examine in this chapter, this is no longer a concern.

## Adding, Removing, and Indexing Elements in an ArrayList

The `ArrayList` class contains a number of methods for adding and removing Objects from the collection. These include `Add`, `AddRange`, `Insert`, `Remove`, `RemoveAt`, `RemoveRange`, and `Clear`, all of which we'll examine in Listing 2.1. The output is shown in Figure 2.1.

### LISTING 2.1 For Sequentially Accessed Collections, Use the ArrayList

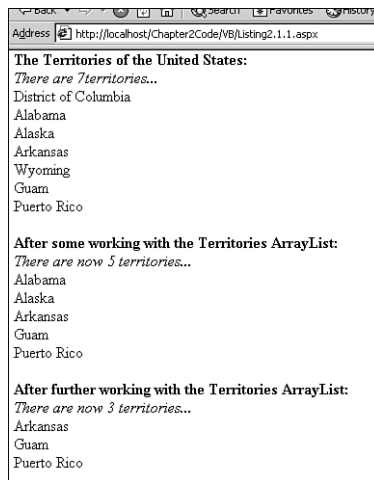
```
1: <script language="vb" runat="server">
2:
3:   Sub Page_Load(source as Object, e as EventArgs)
4:     ' Create two ArrayLists, aTerritories and aStates
5:     Dim aTerritories as New ArrayList
6:     Dim aStates as New ArrayList
7:
8:     ' Use the Add method to add the 50 states of the US
9:     aStates.Add("Alabama")
10:    aStates.Add("Alaska")
11:    aStates.Add("Arkansas")
12:    ' ...
13:    aStates.Add("Wyoming")
14:
15:    ' Build up our list of territories, which includes
16:    ' all 50 states plus some additional countries
17:    aTerritories.AddRange(aStates) ' add all 50 states
18:    aTerritories.Add("Guam")
19:    aTerritories.Add("Puerto Rico")
20:
21:    ' We'd like the first territory to be the District of Columbia,
22:    ' so we'll explicitly add it to the beginning of the ArrayList
23:    aTerritories.Insert(0, "District of Columbia")
24:
25:    ' Display all of the territories with a for loop
26:    lblTerritories.Text = "<i>There are " & aTerritories.Count & _
27:                          "territories...</i><br>"
28:
29:    Dim i as Integer
30:    For i = 0 to aTerritories.Count - 1
31:      lblTerritories.Text = lblTerritories.Text & _
32:                            aTerritories(i) & "<br>"
33:    Next
34:
```

**LISTING 2.1** Continued

```
35:     ' We can remove objects in one of four ways:
36:     ' ... We can remove a specific item
37:     aTerritories.Remove("Wyoming")
38:
39:     ' ... We can remove an element at a specific position
40:     aTerritories.RemoveAt(0) ' will get rid of District
41:                             ' of Columbia,
42:                             ' the first element
43:
44:     ' Display all of the territories with foreach loop
45:     lblFewerTerritories.Text = "<i>There are now " & _
46:         aTerritories.Count & " territories...</i><br>"
47:
48:     Dim s as String
49:     For Each s in aTerritories
50:         lblFewerTerritories.Text = lblFewerTerritories.Text & _
51:             s & "<br>"
52:     Next
53:
54:     ' ... we can remove a chunk of elements from the
55:     ' array with RemoveRange
56:     aTerritories.RemoveRange(0, 2) ' will get rid of the
57:         ' first two elements
58:
59:     ' Display all of the territories with foreach loop
60:     lblEvenFewerTerritories.Text = "<i>There are now " & _
61:         aTerritories.Count & " territories...</i><br>"
62:
63:     For Each s in aTerritories
64:         lblEvenFewerTerritories.Text = lblEvenFewerTerritories.Text & _
65:             s & "<br>"
66:     Next
67:
68:     ' Finally, we can clear the ENTIRE array using the clear method
69:     aTerritories.Clear()
70: End Sub
71:
72: </script>
73:
74: <html>
75: <body>
76:     <b>The Territories of the United States:</b><br>
77:     <asp:label id="lblTerritories" runat="server" />
78:
```

**LISTING 2.1** Continued

```
79: <p>
80:
81: <b>After some working with the Territories ArrayList:</b><br>
82: <asp:label id="lblFewerTerritories" runat="server" />
83:
84: <p>
85:
86: <b>After further working with the Territories ArrayList:</b><br>
87: <asp:label id="lblEvenFewerTerritories" runat="server" />
88: </body>
89: </html>
```

**FIGURE 2.1**

Output of Listing 2.1 when viewed through a browser.

**Adding Elements to an ArrayList**

In Listing 2.1 we create two `ArrayList` class instances, `aTerritories` and `aStates`, on lines 5 and 6, respectively. We then populate the `aStates` `ArrayList` with a small subset of the 50 states of the United States using the `Add` method (lines 9 through 13). The `Add` method takes one parameter, the element to add to the array, which needs to be of type `Object`. This `Object` instance is then appended to the end of the `ArrayList`. In this example we are simply adding elements of type `String` to the `ArrayList` `aStates` and `aTerritories`.

The `Add` method is useful for adding one element at a time to the end of the array, but what if we want to add a number of elements to an `ArrayList` at once? The `ArrayList` class provides

the `AddRange` method to do just this. `AddRange` expects a single parameter that supports the `ICollection` interface. A wide number of .NET Framework classes—such as the `Array`, `ArrayList`, `DataRowView`, `DataSetView`, and others—support this interface. On line 18 in Listing 2.1, we use the `AddRange` method to add each element of the `aStates` `ArrayList` to the end of the `aTerritories` `ArrayList`. (To add a range of elements starting at a specific index in an `ArrayList`, use the `InsertRange` method.) On lines 18 and 19, we add two more strings to the end of the `aTerritories` `ArrayList`.

Because `ArrayLists` are ordered sequentially, there might be times when we want to add an element to a particular position. The `Insert` method of the `ArrayList` class provides this capability, allowing the developer to add an element to a specific spot in the `ArrayList` collection. The `Insert` method takes two parameters: an integer representing the index in which you want to add the new element, and the new element, which needs to be of type `Object`. In line 23 we add a new string to the start of the `aTerritories` `ArrayList`. Note that if we had simply used the `Add` method, "District of Columbia" would have been added to the end of `aTerritories`. Using `Insert`, however, we can specify exactly where in the `ArrayList` this new element should reside.

### Removing Elements from an ArrayList

The `ArrayList` class also provides a number of methods for removing elements. We can remove a specific element from an `ArrayList` with the `Remove` method. On line 37 we remove the String "Wyoming" from the `aTerritories` `ArrayList`. (If you attempt to remove an element that does not exist, an `ArgumentException` exception will be thrown.) `Remove` allows you to take out a particular element from an `ArrayList`; `RemoveAt`, used on line 40, allows the developer to remove an element at a specific position in the `ArrayList`.

Both `Remove` and `RemoveAt` dissect only one element from the `ArrayList` at a time. We can remove a chunk of elements in one fell swoop by using the `RemoveRange` method. This method expects two parameters: an index to start at and a count of total elements to remove. In line 56 we remove the first two elements in `aTerritories` with the statement: `aTerritories.RemoveRange(0, 2)`. Finally, to remove all the contents of an `ArrayList`, use the `Clear` method (refer to Line 69 in Listing 2.1).

### Referencing ArrayList Elements

Note that in our code example, we used two different techniques to iterate through the contents of our `ArrayList`. Because an `ArrayList` stores items sequentially, we can iterate through an `ArrayList` by looping from its lowest bound through its upper bound, referencing each element by its integral index. The following code snippet is taken from lines 30 through 33 in Listing 2.1:



```
For i = 0 to aTerritories.Count - 1
    lblTerritories.Text = lblTerritories.Text & _
        aTerritories(i) & "<br>"
Next
```

The `Count` property returns the number of elements in our `ArrayList`. We start our loop at 0 because all collections are indexed starting at 0. We can reference an `ArrayList` element with: `aArrayListInstance(index)`, as we do on line 32 in Listing 2.1.

We can also step through the elements of any of the collection types we'll be looking at in this chapter using a `For Each ... Next` loop with VB.NET (or a `foreach` loop with C#). A simple example of this approach can be seen in the following code snippet from lines 48 through 52:

```
Dim s as String
For Each s in aTerritories
    lblFewerTerritories.Text = lblFewerTerritories.Text & _
        s & "<br>"
Next
```

This method is useful for stepping through all the elements in a collection. In the future section “*Similarities Among the Collection Types*,” we’ll examine a third way to step through each element of a collection: using an enumerator.

If we wanted to grab a specific element from an `ArrayList`, it would make sense to reference it in the `aArrayListInstance(index)` format. If, however, you are looking for a particular element in the `ArrayList`, you can use the `IndexOf` method to quickly find its index. For example,

```
Dim iPos as Integer
iPos = aTerritories.IndexOf("Illinois")
```

would set `iPos` to the location of Illinois in the `ArrayList` `aTerritories`. (If Illinois did not exist in `aTerritories`, `iPos` would be set to `-1`.) Two other forms of `IndexOf` can be used to specify a range for which to search for an element in the `ArrayList`. For more information on those methods, refer to the .NET Framework SDK documentation.

## Working with the Hashtable Class

The type of collection most developers are used to working with is the hash table collection. Whereas the `ArrayList` indexes each element numerically, a hash table indexes each element by an alphanumeric key. The `Collection` data type in Visual Basic is a hash table; the `Scripting.Dictionary` object, used commonly in classic ASP pages, is a simple hash table. The .NET Framework provides developers with a powerful hash table class, `Hashtable`.

When working with the `Hashtable` class, keep in mind that the ordering of elements in the collection are irrespective of the order in which they are entered. The `Hashtable` class employs its

own hashing algorithm to efficiently order the key/value pairs in the collection. If it is essential that a collection's elements be ordered alphabetically by the value of their keys, use the `SortedList` class, which is discussed in the next section, "Working with the `SortedList` Class."

### Adding, Removing, and Indexing Elements in a Hashtable

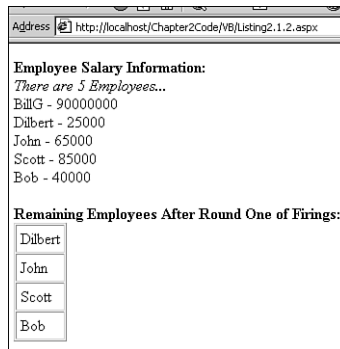
With the `ArrayList` class, there were a number of ways to add various elements to various positions in the `ArrayList`. With the `Hashtable` class, there aren't nearly as many options because there is no sequential ordering of elements. It is recommended that you add new elements to a `Hashtable` using the `Add` method, although you can also add elements implicitly, as we'll see in Listing 2.2. Not surprisingly, there are also fewer methods to remove elements from a `Hashtable`. The `Remove` method dissects a single element from a whereas the `Clear` method removes all elements from a `Hashtable`. Examples of both of these methods can be seen in Listing 2.2. The output is shown in Figure 2.2.

#### LISTING 2.2 For Sequentially Accessed Collections, Use the `ArrayList`

```
1: <script language="VB" runat="server">
2:
3:   Sub Page_Load(source as Object, e as EventArgs)
4:     ' Create a Hashtable
5:     Dim htSalaries As New Hashtable()
6:
7:     ' Use the Add method to add Employee Salary Information
8:     htSalaries.Add("Bob", 40000)
9:     htSalaries.Add("John", 65000)
10:    htSalaries.Add("Dilbert", 25000)
11:    htSalaries.Add("Scott", 85000)
12:    htSalaries.Add("BillG", 90000000)
13:
14:    ' Now, display a list of employees and their salaries
15:    lblSalary.Text = "<i>There are " & htSalaries.Count & _
16:                    " Employees...</i><br>"
17:
18:    Dim s as String
19:    For Each s in htSalaries.Keys
20:      lblSalary.Text &= s & " - " & htSalaries(s) & "<br>"
21:    Next
22:
23:    ' Is BillG an Employee? If so, FIRE HIM!
24:    If htSalaries.ContainsKey("BillG") Then
25:      htSalaries.Remove("BillG")
26:    End If
27:
```

**LISTING 2.2** Continued

```
28:
29:     ' List the remaining employees (using databinding)
30:     dgEmployees.DataSource = htSalaries.Keys
31:     dgEmployees.DataBind()
32:
33:
34:     htSalaries.Clear() ' remove all entries in hash table...
35: End Sub
36:
37: </script>
38:
39: <html>
40: <body>
41:     <b>Employee Salary Information:</b><br>
42:     <asp:label id="lblSalary" runat="server" />
43:     <p>
44:
45:     <b>Remaining Employees After Round One of Firings:</b><br>
46:     <asp:datagrid runat="server" id="dgEmployees"
47:         AutoGenerateColumns="True" ShowHeader="False"
48:         CellSpacing="1" CellPadding="4" />
49: </body>
50: </html>
```

**FIGURE 2.2**

Output of Listing 2.2 when viewed through a browser.

**Adding Elements to a Hashtable**

In Listing 2.2, we begin by creating an instance of the Hashtable class, htSalaries, on line 5. Next, we populate this hash table with our various employees and their respective salaries on

lines 7 through 12. Note that the `Add` method, which adds an element to the `Hashtable` collection, takes two parameters: the first is an alphanumeric key by which the element will be referenced by, and the second is an the element itself, which needs to be of type `Object`.

In Listing 2.2, we are storing integer values in our `Hashtable` class. Of course we are not limited to storing just simple data types; rather, we can store any type of `Object`. As we'll see in an example later in this chapter, we can even create collections of collections (collections whose elements are also collections)!

### Removing Elements from a Hashtable

The `Hashtable` class contains two methods to remove elements: `Remove` and `Clear`. `Remove` expects a single parameter, the alphanumeric key of the element to remove. Line 25 demonstrates this behavior, removing the element referred to as "BillG" in the hash table. On line 34 we remove all the elements of the hash table via the `Clear` method. (Recall that all collection types contain a `Clear` method that demonstrates identical functionality.)

The `Hashtable` class contains two handy methods for determining whether a key or value exists. The first function, `ContainsKey`, takes a single parameter, the alphanumeric key to search for. If the key is found within the hash table, `ContainsKey` returns `True`. If the key is not found, `ContainsKey` returns `False`. In Listing 2.2, this method is used on line 24. The `Hashtable` class also supports a method called `ContainsValue`. This method accepts a single parameter of type `Object` and searches the hash table to see if any element contains that particular value. If it finds such an element, `ContainsValue` will return `True`; otherwise, it will return `False`.

On line 24, a check was made to see if the key "BillG" existed before the `Remove` method was used. Checking to make sure an item exists before removing it is not required. If you use the `Remove` method to try to remove an element that does not exist (for example, if we had `Remove("Homer")` in Listing 2.2), no error or exception will occur. The `ContainsKey` and `ContainsValue` methods are used primarily for quickly determining whether a particular key or element exists in a `Hashtable`.

### The Keys and Values Collections

The `Hashtable` class exposes two collections as properties: `Keys` and `Values`. The `Keys` collection is, as its name suggests, a collection of all the alphanumeric key values in a `Hashtable`. Likewise, the `Values` collection is a collection of all the element values in a `Hashtable`. These two properties can be useful if you are only interested in, say, listing the various keys.

On line 30 in Listing 2.2, the `DataSource` property of the `dgEmployees` `DataGrid` is set to the `Keys` collection of the `hySalaries` `Hashtable` instance. Because the `Keys` property of the `Hashtable` class returns an `ICollection` interface, it can be bound to a `DataGrid` using data binding. For more information on data binding and using the `DataGrid`, refer to Chapter 7, "Data Presentation."

## Working with the SortedList Class

So far we've examined two collections provided by the .NET Framework: the `Hashtable` class and the `ArrayList` class. Each of these collections indexes elements in a different manner. The `ArrayList` indexes each element numerically, whereas the `Hashtable` indexes each element with an alphanumeric key. The `ArrayList` orders each element sequentially, based on its numerical index; the `Hashtable` applies a seemingly random ordering (because the order is determined by a hashing algorithm).

What if you need a collection, though, that allows access to elements by both an alphanumeric key and a numerical index? The .NET Framework includes a class that permits both types of access, the `SortedList` class. This class internally maintains two arrays: a sorted array of the keys and an array of the values.

### Adding, Removing, and Indexing Elements in a SortedList

Because the `SortedList` orders its elements based on the key, there are no methods that insert elements in a particular spot. Rather, similar to the `Hashtable` class, there is only a single method to add elements to the collection: `Add`. However, because the `SortedList` can be indexed by both key and value, the class contains both `Remove` and `RemoveAt` methods. As with all the other collection types, the `SortedList` also contains a `Clear` method that removes all elements.

Because a `SortedList` encapsulates the functionality of both the `Hashtable` and `ArrayList` classes, it's no wonder that the class provides a number of methods to access its elements. As with a `Hashtable`, `SortedList` elements can be accessed via their keys. A `SortedList` that stored `Integer` values could have an element accessed similar to the following:

```
Dim SortedListValue as Integer
SortedListValue = s1SortedListInstance(key)
```

The `SortedList` also can access elements through an integral index, like with the `ArrayList` class. To get the value at a particular index, you can use the `GetByIndex` method as follows:

```
Dim SortedListValue as Integer
SortedListValue = s1SortedListInstance.GetByIndex(iPosition)
```

`iPosition` represents the zero-based ordinal index for the element to retrieve from `s1SortedListInstance`. Additionally, elements can be accessed by index using the `GetValueList` method to return a collection of values, which can then be accessed by index:

```
Dim SortedListValue as Integer
SortedListValue = s1SortedListInstance.GetValueList(iPosition)
```

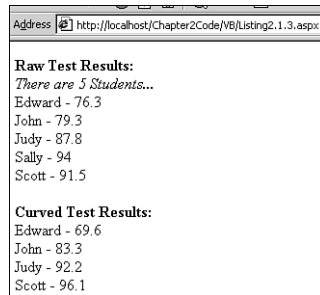
Listing 2.3 illustrates a number of ways to retrieve both the keys and values for elements of a `SortedList`. The output is shown in Figure 2.3.

**LISTING 2.3** A SortedList Combines the Functionality of a Hashtable and ArrayList

```
1: <script language="VB" runat="server">
2:   Sub Page_Load(source as Object, e as EventArgs)
3:     ' Create a SortedList
4:     Dim slTestScores As New SortedList()
5:
6:     ' Use the Add method to add students' Test Scores
7:     slTestScores.Add("Judy", 87.8)
8:     slTestScores.Add("John", 79.3)
9:     slTestScores.Add("Sally", 94.0)
10:    slTestScores.Add("Scott", 91.5)
11:    slTestScores.Add("Edward", 76.3)
12:
13:    ' Display a list of test scores
14:    lblScores.Text = "<i>There are " & slTestScores.Count & _
15:                    " Students...</i><br>"
16:    Dim dictEntry as DictionaryEntry
17:    For Each dictEntry in slTestScores
18:      lblScores.Text &= dictEntry.Key & " - " & dictEntry.Value & "<br>"
19:    Next
20:
21:    'Has Edward taken the test? If so, reduce his grade by 10 points
22:    If slTestScores.ContainsKey("Edward") then
23:      slTestScores("Edward") = slTestScores("Edward") - 10
24:    End If
25:
26:    'Assume Sally Cheated and remove her score from the list
27:    slTestScores.Remove("Sally")
28:
29:    'Grade on the curve - up everyone's score by 5 percent
30:    Dim iLoop as Integer
31:    For iLoop = 0 to slTestScores.Count - 1
32:      slTestScores.GetValueList(iLoop) = _
33:        slTestScores.GetValueList(iLoop) * 1.05
34:    Next
35:
36:    'Display the new grades
37:    For iLoop = 0 to slTestScores.Count - 1
38:      lblCurvedScores.Text &= slTestScores.GetKeyList(iLoop) & " - " & _
39:        Double.Format(slTestScores.GetByIndex(iLoop),
40: "#.##") & "<br>"
41:    Next
42:    slTestScores.Clear() ' remove all entries in the sorted list...
43:  End Sub
```

**LISTING 2.3** Continued

```
44: </script>
45:
46: <html>
47: <body>
48:   <b>Raw Test Results:</b><br>
49:   <asp:label id="lblScores" runat="server" />
50:   <p>
51:
52:   <b>Curved Test Results:</b><br>
53:   <asp:label id="lblCurvedScores" runat="server" />
54: </body>
55: </html>
```

**FIGURE 2.3**

Output of Listing 2.3 when viewed through a browser.

Listing 2.3 begins with the instantiation of the `SortedList` class (line 4). `s1TestScores`, the `SortedList` instance, contains the test scores from five students (see lines 7 through 11). Each element of a `SortedList` is really represented by the `DictionaryEntry` structure. This simple structure contains two public fields: `Key` and `Value`. Starting at line 17, we use a `ForEach ... Next` loop to step through each `DictionaryEntry` element in our `SortedList s1TestScores`. On line 18, we output the `Key` and `Value`, displaying the student's name and test score. Be sure to examine Figure 2.3 and notice that the displayed results are ordered by the value of the key.

On line 22, the `ContainsKey` method is used to see if Edward's score has been recorded; if so, it's reduced by ten points. (Poor Edward.) Note that we access the value of Edward's test score using the element's key—`s1TestScores("Edward")`—just as if `s1TestScores` were a `Hashtable` (line 23). On line 27, Sally's test score is removed from the `SortedList` via the `Remove` method.

Next, each remaining student's test score is upped by 5%. On lines 31 through 34, each test score is visited via a `For ... Next` loop (which is possible because `SortedList` elements can be accessed by an index). Because .NET collections are zero-based, notice that we loop from 0 to `s1TestScores.Count - 1` (line 31). On line 32, the value of each element is accessed via the `GetValueList` method, which returns a collection of values; this collection can then be indexed numerically.

On lines 37 through 40, another `For ... Next` loop is used to display the curved test results. On line 38, the `GetKeyList` method is used to return a collection of keys (which is then accessed by index); on line 39, the test results are outputted using the `Double.Format` function. This format string passed to the `Double.Format` function ("`#.##`") specifies that test results should only display one decimal place. Finally, on line 42, all the test results are erased with a single call to the `Clear` method.

## Working with the Queue Class

`ArrayLists`, `Hashtables`, and `SortedList`s all have one thing in common—they allow random access to their elements. That is, a developer can programmatically read, write, or remove any element in the collection, regardless of its position. However, the `Queue` and `Stack` classes (the remaining two collections we'll examine) are unique in that they provide sequential access only. Specifically, the `Queue` class can only access and remove elements in the order they were inserted.

### Adding, Removing, and Accessing Elements in a Queue

Queues are often referred to as *First In, First Out (FIFO)* data structures because the *N*th element inserted will be the *N*th element removed or accessed. It helps to think of the queue data structure as a line of people. There are two parts to a queue as there are two parts to any line up: the tail of the queue, where people new to the line start waiting, and the head of the queue, where the next person in line waits to be served. In a line, the person who is standing in line first will be first served; the person standing second will be served second, and so on; in a queue, the element that is added first will be the element that is removed or accessed first, whereas the second element added will be the second element removed or accessed.

The .NET Framework provides support for the queue data structure with the `Queue` class. To add an element to the tail, use the `Enqueue` method. To retrieve and remove an element from the head of a queue, use `Dequeue`. As with the other collection types we've examined thus far, the `Queue` class contains a `Clear` method to remove all elements. To simply examine the element at the head without altering the queue, use the `Peek` method. As with all the other collections, the elements of a `Queue` can be iterated through using an enumerator or a `For Each ... Next` loop. Listing 2.4 illustrates some simple queue operations. The output is shown in Figure 2.4.

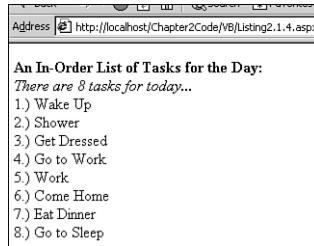


**LISTING 2.4** A Queue Supports First In, First Out Element Access and Removal

```
1: <script language="VB" runat="server">
2:
3:   Sub Page_Load(source as Object, e as EventArgs)
4:     ' Create a Queue
5:     Dim qTasks as New Queue()
6:
7:     qTasks.Enqueue("Wake Up")
8:     qTasks.Enqueue("Shower")
9:     qTasks.Enqueue("Get Dressed")
10:    qTasks.Enqueue("Go to Work")
11:    qTasks.Enqueue("Work")
12:    qTasks.Enqueue("Come Home")
13:    qTasks.Enqueue("Eat Dinner")
14:    qTasks.Enqueue("Go to Sleep")
15:
16:    ' To determine if an element exists in the Queue,
17:    ' use the Contains method
18:    If Not qTasks.Contains("Shower") Then
19:      ' Forgot to bathe!
20:      Response.Write("<b><i>Stinky!</i></b>")
21:    End If
22:
23:    ' Output the list of tasks
24:    lblTaskList.Text &= "<i>There are " & qTasks.Count & _
25:      " tasks for today...</i><br>"
26:
27:    Dim iCount as Integer = 1
28:    Do While qTasks.Count > 0
29:      lblTaskList.Text &= iCount.ToString() & ".) " & _
30:        qTasks.Dequeue() & "<br>"
31:      iCount += 1
32:    Loop
33:
34:
35:    ' At this point the queue is empty, since we've
36:    ' Dequeued all of the elements.
37:  End Sub
38:
39: </script>
40:
41: <html>
42: <body>
43:
44:   <b>An In-Order List of Tasks for the Day:</b><br>
```

**LISTING 2.4** Continued

```
45:     <asp:label runat="server" id="lblTaskList" />
46:
47: </body>
48: </html>
```

**FIGURE 2.4**

Output of Listing 2.4 when viewed through a browser.

**Adding Elements to a Queue**

In Listing 2.4, we begin by creating an instance of the Queue class, `qTasks` (line 5). In line 7 through 14, we add eight new elements to `qTasks` using the `Enqueue` method. Recall that a queue supports First In, First Out ordering, so when we get reader to remove these elements, the first element to be removed will be "Wake Up", which was the first element added.

To quickly check if a particular element is an element of the queue, you can use the `Contains` method. Line 18 demonstrates usage of the `Contains` method. Note that it takes a single parameter, the element to search for, and returns `True` if the element is found in the queue and `False` otherwise.

**Removing Elements from a Queue**

With a Queue, you can only remove the element at the head. With such a constraint, it's no wonder that the Queue class only has a single member to remove an element: `Dequeue`. `Dequeue` not only removes the element at the head of the queue, but it also returns the element just removed.

If you attempt to remove an element from an empty Queue, the `InvalidOperationException` exception will be thrown and you will receive an error in your ASP.NET page. Therefore, to prevent producing a runtime error in your ASP.NET page, be sure to either place the `Dequeue` statement in a `Try ... Catch ... Finally` block or ensure that the `Count` property is greater than zero (0) *before* using `Dequeue`. (For more information on `Try ... Catch ... Finally` blocks, refer to Chapter 9, "ASP.NET Error Handling." For an example of checking the `Count`

property prior to using `Dequeue`, see lines 28 through 32 in Listing 2.4.) As with all the other collection types, you can remove all the `Queue` elements with a single call to the `Clear` method (line 36).

There might be times when you want to access the element at the head of the `Queue` without removing it from the `Queue`. This is possible via the `Peek` method, which returns the element at the head of the `Queue` without removing it. As with the `Dequeue` method, if you try to `Peek` an empty `Queue`, an `InvalidOperationException` exception will be thrown.

### Iterating Through the Elements of a Queue

One way to iterate through the elements of a `Queue` is to simply use `Dequeue` to successively grab each item off the head. This approach can be seen in lines 27 through 32 in Listing 2.4. The major disadvantage of this approach is that, after iteration is complete, the `Queue` is empty!

As with every other collection type, the `Queue` can be iterated via a `For Each ... Next` loop or through the use of an enumerator. The following code snippet illustrates using the C# `foreach` statement to iterate through all the elements of a `Queue` without affecting the structure:

```
Queue qMyQueue = new Queue();    // Create a Queue

qMyQueue.Enqueue(5);
qMyQueue.Enqueue(62);           // Add some elements to the Queue
qMyQueue.Enqueue(-7);

// Iterate through each element of the Queue, displaying it
foreach (int i in qMyQueue)
    Response.Write("Visiting Queue Element with Value: " + i + "<br>");
```

## Working with the Stack Class

A *stack* is a data structure similar to a queue in that it supports only sequential access. However, a stack does bear one major difference from a queue: rather than storing elements with a First In, First Out (FIFO) semantic, a stack uses *Last In, First Out (LIFO)*. A crowded elevator behaves similar to a stack: the first person who enters the crowded elevator is the last person to leave, whereas the last person to board the elevator is the first out when it reaches its destination.

### Adding, Removing, and Accessing Elements in a Stack

The .NET Framework provides an implementation of the stack data type with the `Stack` class. A stack has two basic operations: adding an element to the top of the stack, which is accomplished with the `Push` method, and removing an element from the top of the stack, accomplished via the `Pop` method. Similar to the `Queue` class, the `Stack` class also contains a `Peek` method to permit developers to access the top of the stack without removing the element.

Up until this point, the code provided in the previous listings have just given you a feel for the syntax of the various collections. Listing 2.5, however, contains a handy little piece of reusable code that can be placed on each page of your Web site to provide a set of navigation history links for your visitors.

The code in Listing 2.5 uses a session-level `Stack` class instance that is used to store the links that a Web visitor has traversed on your site since the start of his session. Each time a user visits a Web page, the stack is displayed in a history label and the page's URL is pushed onto the stack. As the user visits various pages on your Web site, his navigation history stack will continue to grow and he will be able to quickly jump back to previous pages on your site. This is, basically, mimicking the functionality of a browser's Back button. The output is shown in Figure 2.5.

---

**LISTING 2.5** A Stack Is Ideal for Keeping Track of a User's Navigation History

---

```
1: <script language="c#" runat="server">
2:
3:   void Page_Load(Object source, EventArgs e)
4:   {
5:       // See if we have a stack created or not:
6:       if (Session["History"] == null)
7:       {
8:           // the history stack has not been created, so create it now.
9:           Session["History"] = new Stack();
10:      } else {
11:          // we already have a history stack. Display the history:
12:          IEnumerator enumHistory =
13:              ((Stack) Session["History"]).GetEnumerator();
14:          while (enumHistory.MoveNext())
15:              lblStackHistory.Text += "<a href=\"\" + enumHistory.Current +
16:                                     \"\">\" + enumHistory.Current +
17:                                     \"</a><br>\";
18:      }
19:
20:      // Push current URL onto Stack IF it is not already on the top
21:      if (((Stack) Session["History"]).Count > 0)
22:      {
23:          if(((Stack) Session["History"]).Peek().ToString() !=
24:              Request.Url.Path.ToString())
25:              ((Stack) Session["History"]).Push(Request.Url.Path);
26:      } else
27:          ((Stack) Session["History"]).Push(Request.Url.Path);
28:  }
29:
```

**LISTING 2.5** Continued

```
30: </script>
31:
32: <html>
33: <body>
34:     <b>Session History</b><br>
35:     <asp:label runat=server id="lblStackHistory" /><br>
36:
37:     <a href="ClearStackHistory.CSharp.aspx">Clear Stack History</a><br>
38:     <a href="Back.CSharp.aspx">Back</a>
39:
40: <p>
41: <b>Links:</b><br>
42: <li><a href="Listing2.1.6.aspx">Listing2.1.6.aspx</a><br>
43: <li><a href="Listing2.1.6.b.aspx">Listing2.1.6.b.aspx</a><br>
44: </p>
45: </html>
```

**FIGURE 2.5**

*Output of Listing 2.5 when viewed through a browser.*

If you've worked with classic ASP, you are likely familiar with the concept of session-level variables. These variables are defined on a per-user basis and last for the duration of the user's visit to the site. These variables are synonymous to global variables in that their values can be accessed across multiple ASP pages. Session-level variables, which are discussed in greater detail in Chapter 14, "Managing State," are a simple way to maintain state on a per-user basis. Because we want the user's navigation history stack to persist as the user bounces around our site, we will store the `Stack` class instance in a session-level variable.

To implement a navigation history stack as a session-level variable, we must make sure that we have created such a variable before trying to reference it. Keep in mind that when a visitor first

comes to our site and visits that first page, the session-level variable will not be instantiated. Therefore, on each page, before we refer to the navigation history stack, it is essential that we check to ensure that our session-variable, `Session["History"]`, has been assigned to an instance of the `Stack` class.

Line 6 in Listing 2.5 checks `Session["History"]` to determine whether it references a `Stack` object instance. If `Session["History"]` has not been assigned an object instance, it will equal `null` (or `Nothing`, in VB). If `Session["History"]` is `null`, we need to set it to a newly created instance of the `Stack` class (line 9).

However, if `Session["History"]` is *not* `null`, we know that the user has already visited at least one other page on our site. Therefore, we can display the contents of the `Session["History"] Stack`. This is accomplished in lines 12 through 17 with the use of an enumerator. We'll discuss iteration through collections via enumerators in the next section, "Similarities Among the Collection Types." With C#, as opposed to VB, explicit casting must be done when working with the `Session` object. For example, on line 13, before we can call the `GetEnumerator()` method (a method of the `Stack` class), we must cast the `Session["History"]` variable to a `Stack`:

```
// C# code must use an explicit cast
IEnumerator enumHistory = ((Stack) Session["History"]).GetEnumerator();
```

```
'VB code, however, does not require an explicit cast
Dim enumHistory As IEnumerator = Session("History").GetEnumerator()
```

With VB, however, such a cast is not necessary. Casting issues with the `Session` object are discussed in more detail in Chapter 14, "Managing State."

After either creating a new session-level `Stack` instance or displaying the `Stack`'s contents, we're ready to add the current URL to the navigation history stack. This could be accomplished with the following simple line of code:

```
((Stack) Session["History"]).Push(Request.Url.Path);
```

However, if the user refreshed the current page, it would, again, get added to the navigation history stack. It would be nice not to have the same page repeatedly appear in the navigation history stack. Therefore, on line 23, we use the `Peek` method to see if the top-most element in the `Stack` is not equal to the current URL; if the top-most element of the stack is not equal to the current URL, we `Push` the current URL onto the top of the stack, otherwise we do nothing.

Before we use the `Peek` method, we first determine whether the `Stack` is empty. Recall from the previous section, "Working with the Queue Class," using the `Peek` method on an empty `Queue` will raise an `InvalidOperationException` exception. This is the same case with the `Stack` class; therefore, on line 21, we first check to ensure that at least one element is in the `Stack` before using the `Peek` method.

Two useful utility ASP.NET pages have been created to provide some extra functionality for our navigation history stack. The first page, `ClearStackHistory.Csharp.aspx`, erases the contents of the history stack and is presented in Listing 2.6. The second page, `Back.Csharp.aspx`, serves like a back button in the user's browser, taking him to the previously visited page. The code for `Back.Csharp.aspx` is given in Listing 2.7.

Listing 2.5 also contains a link to another ASP.NET page, `Listing2.5.b.aspx`. This page is identical to `Listing2.5.aspx`. In your Web site, you would need to, at a minimum, include the code in Listing 2.5 in each ASP.NET page to correctly keep the navigation history up-to-date.

---

**LISTING 2.6** `ClearStackHistory.CSharp.aspx` Erases the Contents of the Navigation History Stack

---

```
1: <script language="c#" runat="server">
2:
3:     void Page_Load(Object source, EventArgs e)
4:     {
5:         // See if we have a stack created or not:
6:         if (Session["History"] == null)
7:         {
8:             // There's no Stack, so we don't need to do anything!
9:         } else {
10:            // we need to clear the stack
11:            ((Stack) Session["History"]).Clear();
12:        }
13:    }
14:
15: </script>
16:
17: <html>
18: <body>
19:     Your navigation history has been cleared!
20: </body>
21: </html>
```

---

Listing 2.6 contains the code for `ClearStackHistory.CSharp.aspx`. This code only has a single task—clear the contents of the navigation history stack—and therefore is fairly straightforward. The ASP.NET page starts by checking to determine if `Session["History"]` refers to a `Stack` object instance (line 6). If it does, the `Clear` method is used to erase all the stack's elements (line 11).

The code for the second utility page, `Back.CSharp.aspx`, can be seen in Listing 2.7.

**LISTING 2.7** Back.CSharp.aspx Sends the User to the Previous Page in His Navigation History Stack

---

```
1: <script language="c#" runat="server">
2:   void Page_Load(Object source, EventArgs e)
3:   {
4:     // See if we have a stack created or not:
5:     if (Session["History"] == null ||
6:         ((Stack) Session["History"]).Count < 2)
7:     {
8:       // There's no Stack, so we can't go back!
9:       Response.Write("Egad, I can't go back!");
10:    } else {
11:      // we need to go back to the prev. page
12:      ((Stack) Session["History"]).Pop();
13:      Page.Navigate(((Stack) Session["History"]).Pop().ToString());
14:    }
15:  }
16: </script>
```

---

As with `ClearStackHistory.CSharp.aspx`, `Back.CSharp.aspx` starts by checking to determine if `Session["History"]` is `null`. If that is the case, a warning message is displayed because we can't possibly step back through our navigation history stack if it doesn't exist!

Take a moment to briefly look over Listing 2.5 again. Note that on each page we visit, we add the current URL to the stack. Therefore, if we want to go back to the previous page, we can't just pluck off the top element from the stack (because that contains the current URL). Rather, we must pluck off the top-most item, dispose of it, and then visit the next item on the top of the stack. For that reason, our stack must have at least two elements to be able to traverse back to the previous page. On line 6, we check to make sure that the navigation history stack contains at least two elements.

Given that we have a properly defined navigation history stack—that is, `Session["History"]` is not `null` and there are at least two elements in the `Stack`—we will reach lines 12 and 13, which do the actual work of sending the user back to the previous page. Line 12 simply disposes of the top-most `Stack` element; line 13 uses the `Navigate` method of the `Page` object to send the user to the next element at the top of the stack.

That wraps up our examination of the navigation history stack example. The code samples spanned three listings: Listing 2.5, Listing 2.6, and Listing 2.7. If you decide to use this code on your Web site, there are a couple of things to keep in mind:

- First, because our implementation of the navigation history stack is a code snippet in an ASP.NET page, the code in Listing 2.5 would need to appear in every Web page on your



site. This, of course, is a ridiculous requirement; it would make sense to encapsulate the code and functionality in a user control to allow for easy code reuse. (For more information on user controls, refer to Chapter 5, “Creating and Using User Controls.”)

- Second, remember that in `Back.CSharp.aspx` we are Popping off the top two URLs. Because Pop removes these elements from the Stack altogether, the navigation history stack cannot contain any sort of Forward link. To provide for both Back and Forward capabilities,

## Similarities Among the Collection Types

Because each collection has the same basic functionality—to serve as a variable-sized storage medium for Objects—it is not surprising that the collection types have much in common with one another. All have methods to add and remove elements from the collection. The `Count` property, which returns the total number of elements in the collection, is common among all collection types.

Each collection also has a means to iterate through each element. This can be accomplished in VB using a `For Each ... Next` loop or, in C#, a `foreach` loop, as follows:

```
'With VB, use a For Each ... Next Loop
Dim qTasks as Queue = New Queue()

' ... Populate the Queue ...

Dim s as String
For Each s in qTasks
    's represents the current element in qTasks
    Response.Write(s + "<br>")
Next

// In C#, a foreach construct can be used to iterate
// through each element
Queue qTasks = new Queue();

// ... Populate the Queue ...

foreach (String s in qTasks)
{
    // s represents the current element in qTasks
    Response.Write(s + "<br>");
}
```

Although each collection can be iterated via a `For Each ... Next` or `foreach` loop, each collection can also have its elements iterated with an enumerator. Enumerators are small classes that provide a simple functionality: to serve as a (read-only) cursor to allow the developer to step through the elements of a collection.

The .NET Framework provides a number of specific enumerators for specific collection types. For example, the `IDictionaryElement` enumerator is useful for iterating through a `Hashtable`. The `IList` enumerator is handy for stepping through the elements of an `ArrayList`. All these specialized enumerators are derived from a base enumerator interface, `IEnumerator`. Because of this fact, all the collection types can be iterated via the `IEnumerator` enumerator as well.

Because an enumerator's most basic purpose is to serve as a cursor for a collection, the `IEnumerator` class contains only a single property that returns the element in the collection to which the enumerator is currently pointing. (More specialized enumerators, such as `IDictionaryElement`, contain multiple properties.) `IEnumerator` contains just two methods: `MoveNext`, which advances the enumerator to the next element in the collection, and `Reset`, which returns the enumerator to its starting position—the position immediately before the first element in the collection.

Listing 2.8 contains a simple ASP.NET page that illustrates iteration through both an `ArrayList` and `Hashtable` with the `IEnumerator` enumerator. The output is shown in Figure 2.6.

---

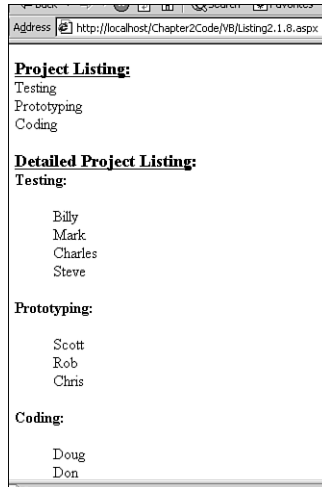
**LISTING 2.8** To Step Through Each Element of a Collection, an Enumerator Can Be Used

---

```
1: <script language="VB" runat="server">
2:
3:   Sub Page_Load(source as Object, e as EventArgs)
4:     ' Create some Collections
5:     Dim aTeam1 as New ArrayList(), _
6:         aTeam2 as New ArrayList(), _
7:         aTeam3 as New ArrayList()
8:
9:     Dim htProjects as New Hashtable()
10:
11:     ' Assign members to the various teams
12:     aTeam1.Add("Scott")
13:     aTeam1.Add("Rob")
14:     aTeam1.Add("Chris")
15:
16:     aTeam2.Add("Doug")
17:     aTeam2.Add("Don")
18:
19:     aTeam3.Add("Billy")
20:     aTeam3.Add("Mark")
21:     aTeam3.Add("Charles")
```

**LISTING 2.8** Continued

```
22:     aTeam3.Add("Steve")
23:
24:
25:     ' Add each team to the htProjects HashTable
26:     htProjects.Add("Prototyping", aTeam1)
27:     htProjects.Add("Coding", aTeam2)
28:     htProjects.Add("Testing", aTeam3)
29:
30:     ' Now, list each project
31:     Dim enumProjects as IEnumerator = htProjects.GetEnumerator()
32:     Do While enumProjects.MoveNext()
33:         lblProjectListing.Text &= enumProjects.Current.Key & "<br>"
34:     Loop
35:
36:     ' Now list each team
37:     Dim enumTeam as IEnumerator
38:     enumProjects.Reset()
39:     Do While enumProjects.MoveNext()
40:         lblDetailedListing.Text &= "<b>" & enumProjects.Current.Key &
":</b><ul>"
41:
42:         enumTeam = enumProjects.Current.Value.GetEnumerator()
43:         Do While enumTeam.MoveNext()
44:             lblDetailedListing.Text &= enumTeam.Current & "<br>"
45:         Loop
46:
47:         lblDetailedListing.Text &= "</ul><p>"
48:     Loop
49: End Sub
50:
51: </script>
52:
53: <html>
54: <body>
55:
56:     <font size=+1><b><u>Project Listing:</u></b></font><br>
57:     <asp:label runat="server" id="lblProjectListing" />
58:     <p>
59:
60:     <font size=+1><b><u>Detailed Project Listing</u></b></font><br>
61:     <asp:label runat="server" id="lblDetailedListing" />
62:
63: </body>
64: </html>
```

**FIGURE 2.6**

*Output of Listing 2.8 when viewed through a browser.*

The code in Listing 2.8 begins by creating three `ArrayList` collections: `aTeam1`, `aTeam2`, and `aTeam3` (lines 5, 6, and 7, respectively). These three `ArrayList`s are then populated with various strings in lines 12 through 22. Each of these `ArrayList`s is added to the `htProjects` `HashTable` on lines 26 through 28. As pointed out earlier, collection types can hold any `Object`, not just simple data types such as integers and strings.

On line 31, an instance of the `IEnumerator` interface is created and assigned to the enumerator for `htProjects`. (Each of the collection types contains a `GetEnumerator()` method that returns a read-only enumerator for the collection.) From lines 32 to 34, the `enumProjects` enumerator is stepped through, visiting each element in the `HashTable` collection.

Note that each element returned to the enumerator from a `HashTable` is an instance of the `DictionaryEntry` object. The `DictionaryEntry` object contains two public fields: `Key` and `Value`. Therefore, on line 33, to obtain the key of the current `HashTable` element, we need to specify that we want the `Key` field of the current element. We could have created a `DictionaryEntry` instance and referenced the `Key` field in a more explicit manner as follows:

```
Dim dictEntry as DictionaryEntry  
Do While enumProjects.MoveNext()
```

```
dictEntry = enumProjects.Current
lblProjectListing.Text &= dictEntry.Key & "<br>"
Loop
```

Because each entry in `htProjects` is an `ArrayList` collection itself, we need to create another enumerator to step through each element in each `ArrayList`. This is accomplished on line 37. At the end of our iteration through `htProjects` in lines 32 through 34, the enumerator `enumProjects` is positioned at the end of the collection. Because we are going to iterate through the `htProjects` collection again, we need to reposition the enumerator back to before the first element. This is accomplished with the `Reset` method of the `IEnumerator` interface (line 38).

In lines 39 through 48, the `htProjects` collection is enumerated through again. This time, each element of `htProjects` is also iterated through itself. On line 42, the `enumTeam` enumerator is assigned via the `GetEnumerator()` method of the current `ArrayList` collection. Next, the `enumTeam` enumerator is stepped through in lines 43 through 45, outputting each `ArrayList` element (line 44).

## Conclusion

The .NET Framework provides developers with a number of powerful collection-type classes, greatly extending the functionality of the `Scripting.Dictionary` object, the sole collection type available for classic ASP developers. These collections, although each have unique capabilities, are more alike than they are different. All of them share similar methods and properties, and can all have their elements iterated through using a number of techniques.

## Working with the File System

Very often Web application developers need to have the ability to access the file system on the Web server. Perhaps they need to list the contents of a particular text file, remove a temporary directory or file, or copy a file from one location to another.

Classic ASP provided adequate support for working with the Web server's file system. The `FileSystemObject` object—along with its accompanying objects such as the `File`, `Folder`, and `TextStream` objects—permitted the classic ASP developer to perform rudimentary tasks with the Web server's file system. One serious shortcoming of the `FileSystemObject` was that the developer, without having to jump through hoops, could only read and write text files; reading and writing binary files with the `FileSystemObject` was possible, but a pain.

The .NET Framework provides a number of classes for working with the file system. These classes are much more robust and have greater functionality than their `FileSystemObject` counterparts. In this section, we'll look at how to accomplish some common file system tasks:

- Reading, creating, and deleting directories with the `Directory` class
- Reading, writing, and creating files

## Reading, Creating, and Deleting Directories

In classic ASP, developers could access directory information with the `Folder` object, one of the many useful `FileSystemObject` objects. The .NET Framework provides a plethora of file system–accessing classes in the `System.IO` namespace, including a `Directory` class. This class will be examined in this section.

Listing 2.9 illustrates the `Directory` class in action! From `Listing2.9.aspx`, the user can enter the name of a directory on the Web server. The page will then list the properties of that directory (if it exists), along with the directory’s subdirectories. The output is shown in Figure 2.7.

### LISTING 2.9 The Directory Class Provides Information About a Particular Directory

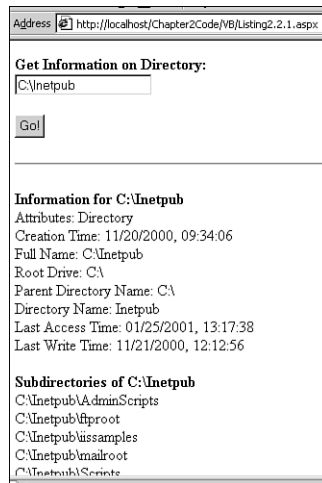
```
1: <%@ Import Namespace="System.IO" %>
2: <script language="VB" runat="server">
3: Sub Page_Load(source as Object, e as EventArgs)
4:   If Not Page.IsPostBack then
5:     lblDirInfo.Text = "Enter the fully qualified name of the " & _
6:       "directory that you're interested in (<i>i.e., C:\</i>)"
7:   Else
8:     ' a postback, so get the directory information
9:     Dim dirInfo as Directory = new Directory(txtDirectoryName.Text)
10:
11:     Try
12:       ' Display the directory properties
13:       lblDirInfo.Text = "<b>Information for " & txtDirectoryName.Text & _
14:         "</b><br> Attributes: " & _
15:         DisplayAttributes(dirInfo.Attributes) & "<br>Creation Time:" & _
16:         dirInfo.CreationTime.ToShortDateString() & _
17:         ", " & dirInfo.CreationTime.ToLongTimeString() & _
18:         "<br>Full Name: " & dirInfo.FullName & "<br>" & _
19:         "Root Drive: " & dirInfo.Root.Name & "<br>" & _
20:         "Parent Directory Name: " & dirInfo.Parent.Name & "<br>" & _
21:         "Directory Name: " & dirInfo.Name & "<br>Last Access Time: " & _
22:         dirInfo.LastAccessTime.ToShortDateString() & ", " & _
23:         dirInfo.LastAccessTime.ToLongTimeString() & "<br>" & _
24:         "Last Write Time: " & dirInfo.LastWriteTime.ToShortDateString() & _
25:         ", " & dirInfo.LastWriteTime.ToLongTimeString() & "<br>"
26:
27:       ' List all of the subdirectories for the current directory:
28:       lblSubDirectories.Text = "<b>Subdirectories of " & _
29:         dirInfo.FullName & "</b><br>"
30:       Dim dirSubDirectory as Directory
```

**LISTING 2.9** Continued

```
31:     For Each dirSubDirectory in dirInfo.GetDirectories()
32:         lblSubDirectories.Text &= dirSubDirectory.FullName & "<br>"
33:     Next
34:     Catch dnfException as DirectoryNotFoundException
35:         ' Whoops! A directoryNotFound Exception has been raised!
36:         ' The user entered an invalid directory name!
37:         lblDirInfo.Text = "<font color=red><b>" & _
38:             dnfException.Message & "</b></font>"
39:     End Try
40: End If
41: End Sub
42:
43:
44: Function DisplayAttributes(fsa as FileSystemAttributes) as String
45: 'Display the file attributes
46: Dim strOutput as String = ""
47:
48: if (fsa BitAnd FileSystemAttributes.Archive) > 0 Then strOutput &=
"Archived, "
49: if (fsa BitAnd FileSystemAttributes.Compressed) > 0 Then strOutput &=
"Compressed, "
50: if (fsa BitAnd FileSystemAttributes.Directory) > 0 Then strOutput &=
"Directory, "
51: if (fsa BitAnd FileSystemAttributes.Encrypted) > 0 Then strOutput &=
"Encrypted, "
52: if (fsa BitAnd FileSystemAttributes.Hidden) > 0 Then strOutput &=
"Hidden, "
53: if (fsa BitAnd FileSystemAttributes.Normal) > 0 Then strOutput &=
"Normal, "
54: if (fsa BitAnd FileSystemAttributes.NotContentIndexed) > 0 Then _
55:     strOutput &= "Not Content
Indexed, "
56: if (fsa BitAnd FileSystemAttributes.Offline) > 0 Then strOutput &=
"Offline, "
57: if (fsa BitAnd FileSystemAttributes.ReadOnly) > 0 Then strOutput &= "Read
Only, "
58: if (fsa BitAnd FileSystemAttributes.ReparsePoint) > 0 Then strOutput &=
"Reparse Point, "
59: if (fsa BitAnd FileSystemAttributes.SparseFile) > 0 Then strOutput &=
"Sparse File, "
60: if (fsa BitAnd FileSystemAttributes.System) > 0 Then strOutput &=
"System, "
61: if (fsa BitAnd FileSystemAttributes.Temporary) > 0 Then strOutput &=
"Temporary, "
62:
```

**LISTING 2.9** Continued

```
63: ' whack off the trailing ", "  
64: If strOutput.Length > 0 Then  
65:     DisplayAttributes = strOutput.Substring(0, strOutput.Length - 2)  
66: Else  
67:     DisplayAttributes = "No attributes found..."  
68: End If  
69: End Function  
70: </script>  
71:  
72: <html>  
73: <body>  
74:     <form method="post" runat="server">  
75:         <b>Get Information on Directory:</b><br>  
76:         <asp:textbox runat="server" id="txtDirectoryName" /><p>  
77:         <asp:button id="btnSubmit" runat="server" type="Submit" text="Go!" />  
78:         <p><hr><p>  
79:         <asp:label runat="server" id="lblDirInfo" /><p>  
80:         <asp:label runat="server" id="lblSubDirectories" />  
81:     </form>  
82: </body>  
83: </html>
```

**FIGURE 2.7**

Output of Listing 2.9 when viewed through a browser.

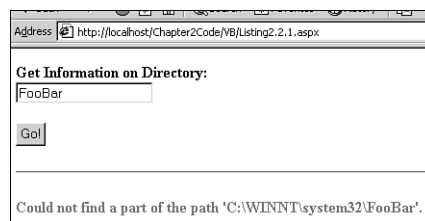


When working with the various file system classes, it is often handy to import the `System.IO` namespace to save unneeded typing (line 1).

Listing 2.9 uses the post-back form technique we discussed in Chapter 1, “Common ASP.NET Page Techniques.” On line 74, a form with the `runat="server"` attribute is created. In the form, there is an `asp:textbox` control and a submit button (`btnSubmit`, line 77). When a user first visits the page, `Page.IsPostBack` is `False` and lines 5 and 6 in the `Page_Load` event handler are executed, displaying an instructional message.

After the user enters a directory name and submits the form, the `Page.IsPostBack` property is set to `True` and the code from lines 8 through 39 is executed. On line 9, a `Directory` object, `dirInfo`, is created. Because the `Directory` class is useful for retrieving information on a particular directory, including the files and subdirectories of a particular directory, it isn’t surprising that the `Directory` constructor expects, as a parameter, the path of the directory with which the developer is interested in working. In this case, we are interested in the directory specified by the user in the `txtDirectoryName` text box.

After we’ve created an instance of the `Directory` class, we can access its methods and properties. However, what if the user specified a directory that does not exist? Such a case would generate an unsightly runtime error. To compensate for this, we use a `Try . . . Catch` block, nesting the calls to the `Directory` classes properties and methods inside the `Try` block (lines 13 through 33). If the directory specified by the user doesn’t exist, a `DirectoryNotFoundException` exception will be thrown. The `Catch` block starting on line 34 will then catch this exception and an error message will be displayed. Figure 2.8 shows the browser output when a user enters a nonexistent directory name.



**FIGURE 2.8**

*An attractive error message is displayed if the user enters an invalid directory name.*

A useful property of the `Directory` class (and the `File` class, which we’ll examine in the next section, “Reading, Writing, and Creating Files”) that deserves further attention is the `Attributes` property. This property is of type `FileSystemAttributes`, an enumeration. The `FileSystemAttributes` enumeration lists the various attributes a directory (or file) can have. Table 2.1 lists these attributes.

**TABLE 2.1** Available Attributes in the `FileSystemAttributes` Enumeration

<i>Attribute</i>	<i>Description</i>
Archive	Indicates the file system entity's archive status.
Compressed	Indicates the file system entity's compression status.
Directory	Indicates if the file system entity is a directory.
Encrypted	Indicates whether the file system entity is encrypted.
Hidden	Indicates if the file system entity is hidden.
Normal	If the file system entity has no other attributes set, it is labeled as Normal.
NotContentIndexed	Indicates whether the file system entity will be indexed by the operating system's indexing service.
Offline	Indicates if the file system entity is offline.
ReadOnly	Indicates whether the file system entity is read-only.
ReparsePoint	Indicates if the file system entity contains a reparse point (a block of user-defined data).
SparseFile	Indicates if a file is defined as a sparse file.
System	Indicates if the file is a system file.
Temporary	Indicates whether the file system entity is temporary or not.

Because each directory (or file) can have a number of attributes (such as a file being both hidden and a system file), the single `Attributes` property has the capability of housing multiple pieces of information. To pick out the individual attributes represented by the `Attributes` property, a bit-wise AND can be used (see lines 48 through 61). To properly display the attributes for a directory in Listing 2.9, a helper function, `DisplayAttributes`, is called from line 15, passing to it the `FileSystemAttributes` enumeration returned by the `Attributes` property.

`DisplayAttributes`, spanning lines 44 through 69, returns a nicely formatted display listing the various attributes indicated by the `FileSystemAttributes` enumeration passed in (`fsa`). On lines 48 through 61, a check is performed to determine if `fsa` contains a particular attribute; if it does, the textual description of the attribute is appended to `strOutput`, which will be returned by `DisplayAttributes` at the end of the function.

The `Directory` class contains two useful methods for retrieving a list of a directory's subdirectories and folders. These two methods are `GetDirectories()`, which returns an array of `Directory` objects representing the subdirectories, and `GetFiles()`, which returns an array of `File` objects representing the list of files in the directory. (We'll examine the `File` object in detail in the next section, "Reading, Writing, and Creating Files." In lines 31 through 33, the

array returned by the `GetDirectories()` method is iterated using a `For Each ... Next` loop, displaying the subdirectories for the directory represented by `dirInfo`.

Listing 2.9 demonstrates how to list the properties of a directory (such as its attributes, creation date, last accessed date, and so on) and how to retrieve the subdirectories for a given directory. However, we have not examined how to create and delete directories.

The `Directory` class contains a number of static methods (methods that can be called without creating a new instance of the `Directory` class). One of these static methods is `CreateDirectory`, which, as its name suggests, creates a directory! Because `CreateDirectory` is a static method, you do not need to create an instance of the `Directory` class to use it. Simply use the following syntax:

```
Directory.CreateDirectory(DirectoryPath)
```

The `CreateDirectory` method will throw an `IOException` exception if the directory *DirectoryPath* already exists. Keep in mind that `CreateDirectory` will create only one directory. For example, imagine that we want to create a directory `C:\ASP\CodeExamples`. We might try

```
Directory.CreateDirectory("C:\ASP\CodeExamples")
```

However, if the directory `C:\ASP` does not exist, the `CreateDirectory` method will throw a `DirectoryNotFoundException` exception because `CreateDirectory` can't create the `CodeExamples` directory because the `C:\ASP` directory does not exist! Fortunately, the .NET Framework contains another static method (`CreateDirectories`), which can create multiple directories if needed. The following code will work regardless of whether the `C:\ASP` directory currently exists:

```
Directory.CreateDirectories("C:\ASP\CodeExamples")
```

To delete a directory, use the `Delete` method. The `Delete` method has a static version that has two forms:

```
Directory.Delete(DirectoryPath)  
Directory.Delete(DirectoryPath, RecurseDirs)
```

The *DirectoryPath* is the path of the directory that you want to delete. *RecurseDirs* is a Boolean value, which if `True`, will delete the directory, all its files, and all its subdirectories and their files. If *RecurseDir* is `False` (or not specified at all) and you attempt to delete a directory that contains any files or subdirectories, an `IOException` exception will be thrown.

There is also a non-static version of the `Delete` method. Because when creating an instance of the `Directory` class, you must specify a directory path in the constructor, there is no need for the nonstatic version to require a *DirectoryPath* parameter. So, to delete the directory `C:\ASP`, either one of these approaches will work:

```
'Delete C:\ASP with the static Delete method
Directory.Delete("C:\ASP", True)

'Delete C:\ASP with the non-static Delete method
'First create an instance of the Directory class
Dim dirASP as New Directory("C:\ASP")
dirASP.Delete(True)
```

**WARNING**

When working with the file system using C#, keep in mind that the string escape sequence for C# is the backslash (\). To insert a literal backspace into a string, you must use two consecutive backspaces. For example, to delete a directory, use `Directory.Delete("C:\\ASP");`

## Reading, Writing, and Creating Files

Because the .NET Framework provides a class for retrieving information about a particular directory (the `Directory` class), it should come as no surprise that it also provides a class for accessing file information. This class, aptly named `File`, contains a number of properties similar to the `Directory` class. For example, the `Attributes`, `CreationTime`, `Exists`, `FullName`, `LastAccessedTime`, `LastWriteTime`, and `Name` properties are common to both the `File` and `Directory` classes.

The methods of the `File` class are fairly straightforward; they provide the basic functionality for files. The methods to open a file are `Open`, `OpenRead`, `OpenText`, and `OpenWrite`. The methods to create a file are `Create` and `CreateText`. The methods to delete and do miscellaneous file-related tasks are `Copy`, `Delete`, `ChangeExtension`, and `Move`.

Listing 2.10 illustrates how to read (and display) the contents of a text file, as well as how to use a `DataList` and databinding to display the contents of an array. A thorough examination of databinding and use of the `DataList` can be found in Chapter 7, “Data Presentation.”

---

**LISTING 2.10** The `File` Class Can Be Used to Retrieve Properties or the Contents of a File on the Web Server

---

```
1: <%@ Import Namespace="System.IO" %>
2: <script language="VB" runat="server">
3: Sub Page_Load(source as Object, e as EventArgs)
4:   If Not Page.IsPostBack then
5:     ' What directory are we interested in?
6:     const strDir = "C:\My Projects\ASP.NET Book\Chapter 2\Code\VB"
```

**LISTING 2.10** Continued

```
7:     lblHeader.Text = "<b><u>File Listing for " & strDir & " :</u></b>"
8:
9:     ' Get the files for the directory strDir
10:    Dim aFiles as File() = Directory.GetFilesInDirectory(strDir, "*.aspx")
11:
12:    dlFileList.DataSource = aFiles
13:    dlFileList.DataBind()
14:  End If
15: End Sub
16:
17:
18: Sub dlFileList_Select(sender as Object, e as EventArgs)
19:   Dim strFilePath as String = _
20:     dlFileList.DataKeys(dlFileList.SelectedItem.ItemIndex).ToString()
21:   Dim objFile as File = new File(strFilePath)
22:
23:   Dim objStream as StreamReader = objFile.OpenText()
24:   Dim strContents as String = objStream.ReadToEnd()
25:   objStream.Close()
26:
27:   lblFileContents.Text = "<b>Contents of " & objFile.Name & " :</b>" & _
28:     "<xmp>" & vbCrLf & strContents & vbCrLf & "</xmp>"
29: End Sub
30: </script>
31:
32: <html>
33: <body>
34:   <form runat="server">
35:     <asp:label id="lblHeader" runat="server" /><br>
36:     <asp:DataList runat="server" id="dlFileList"
37:       OnSelectedIndexChanged="dlFileList_Select"
38:       DataKeyField="FullName" >
39:       <template name="itemtemplate">
40:         <li><%# DataBinder.Eval(Container.DataItem, "Name") %><br>
41:         <font size=-1>
42:           [<asp:linkbutton Text="View Contents"
43:             CommandName="Select" runat="server"/>] |
44:           [<%# DataBinder.Eval(Container.DataItem, "Length") %> bytes]
45:         </font>
46:         <p>
47:       </template>
48:     </asp:DataList>
49:     <p><hr><p>
50:     <asp:label runat="server" id="lblFileContents" />
```

**LISTING 2.10** Continued

---

```
51:     </form>
52: </body>
53: </html>
```

---

The code in Listing 2.10 serves a very simple purpose: to list the ASP.NET pages in a particular directory and to allow the user to view the source code for any one of these pages. This can be thought of as two separate tasks:

1. Listing the files in a particular directory
2. Displaying the contents of the selected file

The first task is handled by the `Page_Load` event handler (lines 3 through 15) and the `DataList` control (lines 36 through 48). The first line of the `Page_Load` event handler checks to determine if the page is being visited for the first time (if so, `Page.IsPostBack` will be `False`, and the code from lines 5 through 13 will be executed). In such a case, we want to display the files for a particular directory. On line 6, the directory path whose files will be displayed has been hard coded. By using concepts from Listing 2.9, however, Listing 2.10 could be expanded to allow the user to specify the directory.

Next, those files in the directory `strDir` that end with the `.aspx` extension are returned (line 10). The `GetFilesInDirectory` method of the `Directory` class is a static method that can accept one or two parameters. The first parameter is required and is the path of the directory whose file listing to return. The second, optional parameter is a search criteria field in which wildcards can be used to limit the files returned. Because we are only interested in listing ASP.NET pages, we want to grab only those files that have the `.aspx` extension. The `GetFilesInDirectory` method returns an array of `File` objects, which we assign to our variable `aFiles` (line 10).

On lines 12 and 13, we bind this array to `d1FileList`, our `DataList` whose definition begins on line 36. The `DataList` uses databinding syntax to display the `Name` property of each `File` object in the `aFiles` array (line 40) along with the `Length` property, which indicates the file's size in bytes (line 44). In the `DataList` heading (lines 36 through 38), the `SelectedIndexChanged` event is wired up to the `d1FileList_Select` event handler; furthermore, the `DataList` specifies the `FullName` property of the `File` class as its `DataKeyField` (line 38).

A `LinkButton` server control is created on lines 42 and 43 with a `CommandName` of `Select`. When this `LinkButton` is clicked, the page will be reposted and the `d1FileList_Select` event handler will be called. From the `d1FileList_Select` event handler, the `FullName` of the file clicked can be programmatically determined because of the `DataKeyField` property on line 38.

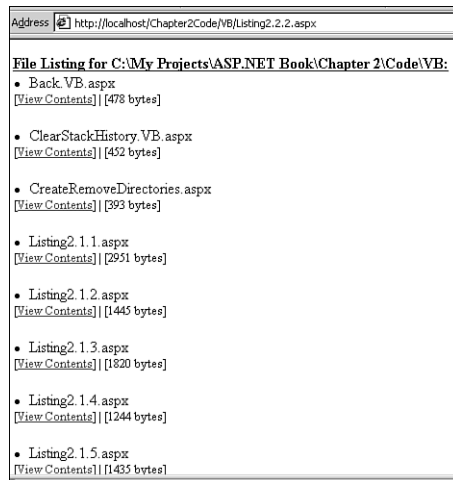
If you are unfamiliar with the `DataList` control and databinding, this might all be a bit overwhelming to you. Don't worry, though, databinding will be discussed thoroughly in Chapter 7, "Data Presentation."

After a View Contents link is clicked, the page will be reloaded and we're on to the second task: displaying the contents of the selected file. This is handled in the `d1FileList_Select` event handler. On line 19, the clicked `LinkButton`'s `DataKeyField` is extracted and stored in the variable `strFilePath`. This contains the full path to the file that we want to display.

Next, on line 21, a `File` object is instantiated and the constructor is called, passing it the path of the file we are interested in. To simply read the contents of a text file, the `File` class provides an `OpenText` method, which returns a `StreamReader` instance that can be used to step through the contents of the file. On line 23, a `StreamReader` instance is created and assigned to the object returned by the `OpenText` method. Next, the entire stream is read into a string variable, `strContents` (line 24), and the stream is closed (line 25).

On line 27 and 28, the contents of the selected file are displayed in the `lb1FileContents` label server control. Because we are displaying the contents of a file that likely contains HTML and script-block code, the contents are surrounded by a pair of `XMP` tags. (The `XMP` tag is a standard HTML tag that displays text ignoring all HTML tags.)

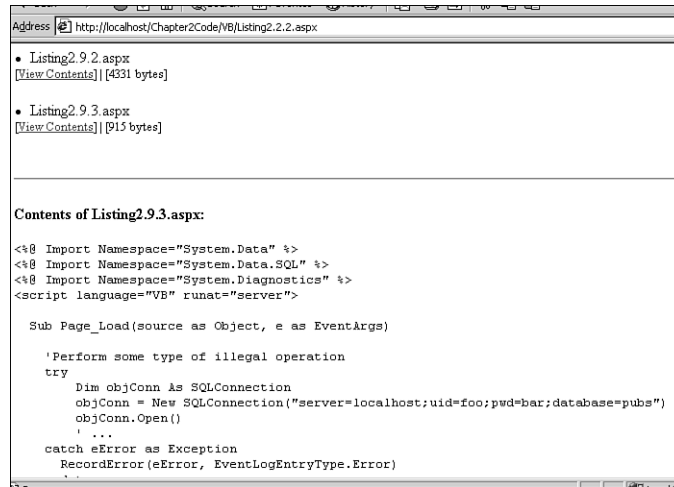
Figure 2.9 shows the output of the code in Listing 2.10 when we first visit the page. Note that the contents of the specified directory are displayed with a link to view the source and a note on their file size.



**FIGURE 2.9**

*When the user first visits the page, she is shown a listing of files.*

When the user clicks the View Contents link for a particular file, the page is reloaded and the file's contents are displayed beneath the file listing. Figure 2.10 shows what a user would see after clicking on a particular file in the listing.



**FIGURE 2.10**

*Clicking on a particular file displays its contents beneath the file listing.*

In Listing 2.10 we examined how to list the contents of a directory, how to list certain File class properties, and how to read the contents of a text file. However, we've yet to look at how to create a file and how to work with binary files (a task that was difficult in classic ASP with the `FileSystemObject`).

## Creating Text Files

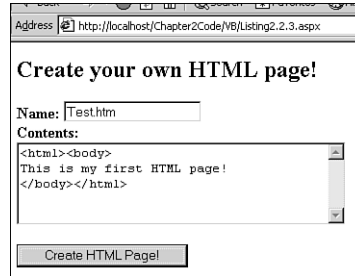
Let's first examine how to create a file. Listing 2.11 demonstrates an ASP.NET page that serves as a very basic Create your own Web Page utility. From the page, users can enter a filename for their HTML page and the HTML code; when they click the Create HTML Page button, a new HTML page will be created on the server with the HTML syntax they entered. After this file is created, anyone can view it via his or her browser.

The script in Listing 2.11 demonstrates the functionality of an extremely simple HTML file editor that might be found on a site that allows users to create personal home pages. Listing 2.11 is written in C#; note some of the similarities and differences between C#'s syntax and VB's syntax. For example, to insert a literal backslash into a string, when using C# you must use two consecutive backslashes (line 6). The output is shown in Figure 2.11.



**LISTING 2.11** The File and StreamWriter Classes Allow Developers to Create Files on the Web Server

```
1: <%@ Import Namespace="System.IO" %>
2: <script language="C#" runat="server">
3:     void btnSubmit_OnClick(Object source, EventArgs e)
4:     {
5:         // Create an HTML file in the directory strDir
6:         const String strDir = "C:\\Inetpub\\wwwroot\\UserPages\\";
7:
8:         String strFileName = strDir + txtPageName.Text;
9:
10:        // Create the file using the static method CreateText
11:        StreamWriter objStream = File.CreateText(strFileName);
12:
13:        // Write the contents of the txtContents textarea to the stream
14:        objStream.Write(txtContents.Text);
15:
16:        // Close the stream, saving the file...
17:        objStream.Close();
18:
19:        // Display a link to the newly created HTML page.
20:        lblLink.Text = "Page created!<br><a href=\\\"/UserPages/\" +
21:            txtPageName.Text + "\\\">View New Page!</a>";
22:    }
23: </script>
24:
25: <html>
26: <body>
27:     <form runat="server">
28:         <font size=+2><b>Create your own HTML page!</b></font>
29:         <p>
30:         <b>Name:</b> <asp:textbox id="txtPageName" runat="server" /><br>
31:         <b>Contents:</b><br>
32:         <asp:textbox id="txtContents" runat="server" TextMode="MultiLine"
33:             Columns="40" Rows="5" />
34:         <p>
35:         <asp:button id="btnSubmit" runat="server" text="Create HTML Page!"
36:             OnClick="btnSubmit_OnClick" />
37:         <p>
38:         <asp:label id="lblLink" runat="server" />
39:     </form>
40: </body>
41: </html>
```



The screenshot shows a web browser window with the address bar displaying `http://localhost/Chapter2Code/WB/Listing2.2.3.aspx`. The main content area has the heading "Create your own HTML page!". Below the heading is a form with two input fields and a button. The first field is labeled "Name:" and contains the text "Test.htm". The second field is labeled "Contents:" and contains the HTML code: `<html><body>This is my first HTML page!</body></html>`. At the bottom of the form is a button labeled "Create HTML Page!".

**FIGURE 2.11**

*The user can enter HTML and a filename and the ASP.NET page will generate a file on the Web server with the proper name and contents!*

Listing 2.11 is the first code sample we've looked at in this chapter that did not include the `Page_Load` event handler. Therefore, before looking at the C# code, let's turn our attention to the HTML portion first (lines 25 through 41). First, note that we are using post-back forms, so we must create a form with the `runat="server"` attribute (line 27). Next, we need two text boxes: one for the user to enter the filename of the HTML page he wants to create and another for the HTML contents for the page.

The text box for the filename, `txtPageName`, is created on line 30. The text box for the HTML contents, `txtContents`, is created as a multiline text box with 40 columns and 5 rows (lines 32 and 33). A multiline text box is, in HTML terms, a `TEXTAREA`. Next, we create a button and wire up its click event to the `btnSubmit_OnClick` event handler (lines 35 and 36). The last server control is a label, `lblLink`, which we'll use to display a link to the newly created HTML page.

When the Create HTML Page! button is clicked, the post-back form is submitted and the `btnSubmit_OnClick` event handler is called. This event handler's task is to create the file specified by the user with the HTML contents specified by the user. All the HTML files created by the user will be placed in the directory specified by the string constant `strDir` (line 6). The actual filename for the file to be created is the concatenation of `strDir` and the value the user entered in the `txtPageName` text box (line 8).

On line 11 the static version of the `CreateText` method is used to create a new text file. The static version of the `CreateText` method expects a single parameter, the path to the file to create, and returns a `StreamWriter` object that can be used to write the contents to the newly created file. On line 14, the `Write` method of the `StreamWriter` class is used to write the HTML contents entered by the user to the file. This file is saved on line 17, when the `Close` method is called. Finally, a link to the newly created file is displayed on lines 20 and 21.

**WARNING**

The code in Listing 2.11 does not contain any sort of error-handling code. If the user doesn't enter a filename, an exception will be thrown. This could be remedied via a try ... catch block, an if statement in btnSubmit\_OnClick to check if txtPageName contains a valid value, or through a validation control. (For more information on try ... catch blocks, be sure to read Chapter 9, "ASP.NET Error Handling." For information on validation controls, refer to Chapter 3, "Form Field Input Validation.")

## Working with Binary Files

The `FileSystemObject` objects used with classic ASP were designed to work with text files. The file system objects in the .NET Framework, however, were designed to be much more flexible, able to easily work with binary or text files alike. So far, we've examined how to read and create text files. It's time that we took a quick look at how binary files are handled.

The code in Listing 2.12 uses the static version of the `Open` method of the `File` class to open a binary file (in this case, a GIF file). The binary data is then squirted to the user's browser via the `Response.BinaryWrite` method. Because browsers can inherently display GIF files, the user ends up seeing the image file as though he had directed his browser directly to the image file on the Web server (as opposed to an ASP.NET page).

### LISTING 2.12 The .NET Framework Can Easily Handle Both Binary and Text Files

```
1: <%@ Import Namespace="System.IO" %>
2: <script language="vb" runat="server">
3:   Sub Page_Load(source as Object, e as EventArgs)
4:     Const strFileName as String = "C:\Inetpub\wwwroot\Web.gif"
5:
6:     ' Read the contents of a binary file
7:     Dim objStream as Stream = File.Open(strFileName, FileMode.Open)
8:
9:     Dim buffer(objStream.Length) as Byte
10:    objStream.Read(buffer, 0, objStream.Length)
11:    objStream.Close()
12:
13:    Response.BinaryWrite(buffer)
14:  End Sub
15: </script>
```

Listing 2.12 displays a GIF in the users browser specified by the hard-coded file path on line 4. Next, on line 7, the `Open` method is used to retrieve a `Stream` object to the GIF file's contents.

There are many variations of the `Open` method; in this example we pass the `Open` method two parameters: the path to the file to open and the `FileMode` access to use. `FileMode` is an enumeration with its various entries representing various modes of which a file can be accessed. Table 2.2 lists these various file modes.

**TABLE 2.2** The `FileMode` Enumeration Contains the Various Modes that a File Can Be Accessed

<i>Attribute</i>	<i>Description</i>
Append	If the file exists, it is opened and the stream is positioned at the end of the file; if the file does not exist, it is created.
Create	Creates a new file if the file does not exist; if the file does exist, it is overwritten.
CreateNew	Specifies that a new file should be created.
Open	Opens an existing file.
OpenOrCreate	Specifies that the file should be opened if it exists, and created if it does not currently exist.
Truncate	Opens an existing file and positions the stream at the beginning of the file to overwrite existing data.

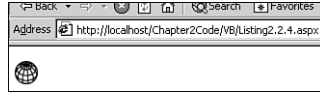
The `Open` method returns a `Stream` object that we can use to read from the file. The `Stream` class contains a `Read` method that takes three parameters: a buffer as a byte array, an integral offset, and an integral count, in the following format:

```
File.Read(buffer(), offset, count)
```

The `Read` method will then dump `count` bytes from the stream into the `buffer` array starting at a specified `offset` in the array. Before we execute this statement, though, we need to create and initialize a buffer array. Because we want to read the entire contents of the GIF file into our array, we need to create an array the size of the GIF file. This size can be retrieved from the `Length` property of the `Stream` class (line 9).

When we have this array properly initialized, we can go ahead and dump all the contents of the GIF file into the buffer using the `Read` method (line 10). Note that we are beginning the dump into the start of the buffer (hence `offset` is set to zero) and we are reading the entire stream (hence `count` is set to the length of the stream—`objStream.Length`).

Finally, the byte array is squirted to the browser using the `BinaryWrite` method of the `Response` object. The code in Listing 2.12, when viewed through a browser, will display the GIF specified by `strFileName` (line 4). Figure 2.12 is a screenshot of the browser visiting `Listing2.12.aspx`.

**FIGURE 2.12**

*Use the Stream object to read binary files.*

## Using Regular Expressions

Regular expressions are a neat and efficient way to perform extremely powerful string pattern matching and replacing. For example, imagine that you had an HTML document and you wanted to pick through it to retrieve all the text between any bold tags (between `<b>` and `</b>`). Although this can be done with `String` class methods such as `IndexOf` and `Substring`, it would involve an unnecessarily complex loop with some ugly code. Instead, a relatively simple regular expression could be used.

Originally with classic ASP, regular expression support was a language-specific feature. That is, it was up to the syntax language being used to support regular expressions. With the release of the Microsoft Scripting Engine Version 5.0, Microsoft released a regular expression COM component to handle regular expression support in the same manner regardless of the scripting language.

With the .NET Framework, regular expressions are supported via a number of classes in the `System.Text.RegularExpressions` namespace. In this section we will examine these classes and look at some code samples. This section is not intended to teach regular expressions fundamentals—rather, it aims to illustrate how to work with regular expressions using the classes in the `System.Text.RegularExpressions` namespace. For some very useful, very handy, real-world regular expressions, be sure to check out the next chapter, Chapter 3, “Form Field Input Validation,” which has a section on common regular expression validations for the `RegularExpressionValidator` validation control.

The class in the `System.Text.RegularExpressions` namespace that handles the bulk of the regular expression work is the `Regex` class. The constructor for this class is very important because it requires the most essential part of a regular expression: the pattern. Two forms of the constructor are as follows:

```
'Requires the string pattern  
Regex(pattern)
```

```
'Requires the string pattern and regular expression options  
Regex(pattern, options)
```

Note that in both cases, *pattern*, which is of type `String`, is required. *options*, if specified, needs to be a string of single character options.

The `Regex` class contains a number of methods for finding matches to a pattern in a string, replacing instances of the matching pattern with another string and testing to see if a matching pattern exists in a string. Let's look at the `IsMatch` method, which tests to see if a pattern is found in a string.

There are both static and non-static versions of the `IsMatch` method. The non-static version, which requires a `Regex` instance, requires only one parameter, the string to search for the pattern. (Recall that you must supply the pattern for the regular expression in the constructor.) A very simple `IsMatch` example can be seen in Listing 2.13.

---

**LISTING 2.13** The `IsMatch` Method Determines if a Pattern Is Found in a String

---

```
1: <%@ Import Namespace="System.Text.RegularExpressions" %>
2: <script language="VB" runat="server">
3: Sub Page_Load(source as Object, e as EventArgs)
4:   Dim str as String = "Reality, the external world, exists " & _
5:     "independent of man's consciousness, independent of any observer's " &
6:     "knowledge, beliefs, feelings, desires or fears. This means that A is
A..."
7:
8:   ' Check to see if the string str contains the pattern 'A is A'
9:   Dim regexp as Regex = new Regex("A is A", "i")
10:
11:   If regexp.IsMatch(str) then
12:     Response.Write("This is an Ayn Rand quote.")
13:   Else
14:     Response.Write("I don't know who said this.")
15:   End If
16: End Sub
17: </script>
```

---

Because the `Regex` class exists in the `System.Text.RegularExpressions` namespace, our first line of code imports the proper namespace so that we can refer to the `Regex` class without fully qualifying it (line 1). On lines 4 through 6, a string, `str`, is hard-coded with a quote from Ayn Rand. Next, on line 9, an instance of the `Regex` class is created. This instance, `regexp`, is created with the `Regex` constructor that takes two parameters, the pattern and options strings. The pattern `'A is A'` will simply match the substring `"A is A"`; the option `"i"` indicates that the search should not be case sensitive.

On line 11, the `IsMatch` method is used to check if the substring "A is A" exists in the string `str` (line 11). `IsMatch` returns a Boolean value: `True` if the pattern is found in the passed-in string, `False` otherwise. If the substring "A is A" is found in `str`, "This is an Ayn Rand quote." is displayed; otherwise "I don't know who said this." is displayed. As you might have guessed, the output of Listing 2.14, when viewed through a browser, is "This is an Ayn Rand quote.".

As mentioned earlier, there is also a static version of the `IsMatch` method. The static version takes either two or three parameters. The first parameter is the input string, the second is the regular expression pattern, and the third option parameter is the options string for the regular expression. In Listing 2.14, line 9 could be snipped and line 11 replaced with the following:

```
If Regex.IsMatch(str, "A is A", "i") then
```

Finding out whether a regular expression pattern exists in a string is all well and good, but being able to grab a listing of substrings that matched would be ideal. The `Matches` method of the `Regex` has such functionality. The non-static version of the method expects a single parameter, the string to search, and returns the resulting matches as a `MatchCollection`.

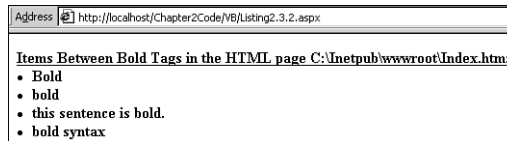
Listing 2.15 uses the `Matches` method to list all the text between the bold tags in an HTML document. This code borrows some file-reading code from Listing 2.10 to read in the contents of a text file on the Web server. The output is shown in Figure 2.13.

### LISTING 2.15 The `Matches` Method Will Return All the Matching Regular Expression Patterns in a String

```
1: <%@ Import Namespace="System.IO" %>
2: <%@ Import Namespace="System.Text.RegularExpressions" %>
3: <script language="VB" runat="server">
4:   Sub Page_Load(source as Object, e as EventArgs)
5:     'Read in the contents of the file strFilePath
6:     Dim strFilePath as String = "C:\Inetpub\wwwroot\Index.htm"
7:     Dim objFile as File = new File(strFilePath)
8:     Dim objStream as StreamReader = objFile.OpenText()
9:     Dim strContents as String = objStream.ReadToEnd()
10:    objStream.Close()
11:
12:    'List the text between the bold tags:
13:    Dim regexp as Regex = New Regex("<b>((.|\\n)*?)</b>", "i")
14:
15:    Response.Write("<u><b>Items Between Bold Tags in the HTML page " & _
16:                  strFilePath & " :</b><</u><br>")
17:
18:    'Create a Match object instance / iterate through the MatchCollection
```

**LISTING 2.15** Continued

```
19: Dim objMatch as Match
20: For Each objMatch in regexp.Matches(strContents)
21:     Response.Write("<li>" & objMatch.ToString() & "<br>")
22: Next
23: End Sub
24: </script>
```

**FIGURE 2.13**

The HTML text between (and including) each bold tag pair is returned as its own Match object.

Listing 2.15 begins with two `Import` directives: the first imports the `System.IO` namespace to assist with our use of the `File` class (line 1); the second imports the `System.Text.RegularExpressions` namespace to assist with our use of the `Regex` and `Match` classes (line 2).

When the `Imports` are out of the way, Listing 2.15 starts by opening and reading the contents of a hard coded HTML file (lines 6 through 10). The contents of this HTML file are stored in the variable `strContents`. (This code snippet should look familiar—it’s taken directly from Listing 2.10!)

Next, a regular expression instance is created with a pattern set to `<b>((.|\n)*?)</b>`. This might seem a bit confusing, especially if you are not very familiar with regular expressions. Translated to English, the pattern would read: “Find a bold tag (`<b>`), find zero or more characters, and then find a closing bold tag (`</b>`).” The period (`.`) is a special character in regular expressions, meaning, “Match any character except the new line character.” The new line character is represented by `\n`; the asterisk (`*`) means to match zero or more characters, and the question mark following the asterisk means to perform “non-greedy” matching. For a more thorough explanation of all these terms, be sure to read *Picking Out Delimited Text with Regular Expressions*, available at <http://www.4guysfromrolla.com/webtech/103100-1.shtml>.

When we have our regular expression object instance, we’re ready to call the `Matches` method. The `Matches` method returns a `MatchCollection` class instance, which is, essentially, a collection of `Match` objects. The `Match` class contains various properties and methods that provide information on a particular match of a regular expression pattern in a string.



To iterate through the `MatchCollection`, we first create a `Match` instance, `objMatch`, on line 19. Next, a `For Each ... Next` loop is used to iterate through each resulting `Match` returned by the `Matches` method on the HTML contents of the file (line 20). On line 21, the matched text is outputted. Figure 2.13 shows the output of Listing 2.15 when the file `C:\inetpub\wwwroot\Index.htm` contains the following text:

```
<html>
<head><title>Hello, World!</title></head>
<body bgcolor=white text=black>

    <b>Bold</b> text in HTML is defined using the <b>bold</b> tags:
    <code>&lt;b&gt; ... &lt;/b&gt;</code>. For example, <b>this
    sentence is bold.</b> This sentence is not bold. <i>This sentence
    is in italics!</i>
    <p>
    To learn more about the <b>bold syntax</b>, read a book covering
    HTML syntax in-depth!

</body>
</html>
```

Another useful task of regular expressions are their capability of doing powerful string replacements. For example, many Web sites have a searching feature in which the user enters a keyword to search on and various results are returned. Wouldn't it be nice to have the words in the results that matched the search keyword to be highlighted?

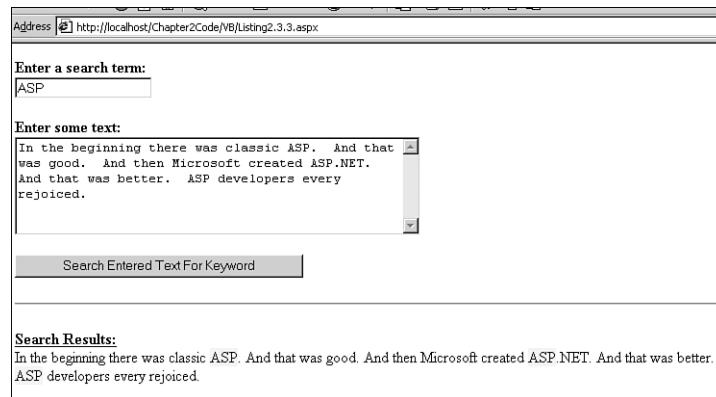
Listing 2.16 contains the code for an ASP.NET page that allows the user to enter both a search term and some text that the search term will be applied to. Any instances of the search term are highlighted. The output is shown in Figure 2.14.

### LISTING 2.16 Regular Expressions Perform Search and Replace Features on Complex Patterns

```
1: <%@ Import Namespace="System.Text.RegularExpressions" %>
2: <script language="VB" runat="server">
3:   Sub btnSubmit_OnClick(Source as Object, e as EventArgs)
4:     'Create a regex object
5:     Dim strTerms as String = txtSearchTerms.Text
6:     Dim regexp as Regex = new Regex("\b" & strTerms & "\b", "i")
7:
8:     'Replace all search terms in txtText
9:     Dim strNewText as String = regexp.Replace(txtText.Text, _
10:        "<span style='color:black;background-color:yellow'>" & strTerms &
"</span>")
11:
```

**LISTING 2.16** Continued

```
12:     lblResults.Text = "<p><hr><p><b><u>Search Results:</u></b><br>" &
strNewText
13: End Sub
14: </script>
15:
16: <html>
17: <body>
18:     <form runat="server">
19:         <b>Enter a search term:</b><br>
20:         <asp:textbox id="txtSearchTerms" runat="server" />
21:         <p>
22:         <b>Enter some text:</b><br>
23:         <asp:textbox id="txtText" runat="server" TextMode="MultiLine"
24:             Cols="50" Rows="6" />
25:         <p>
26:         <asp:button id="btnSubmit" runat="server" OnClick="btnSubmit_OnClick"
27:             text="Search Entered Text For Keyword" />
28:         <p><asp:label id="lblResults" runat="server" />
29:     </form>
30: </body>
31: </html>
```

**FIGURE 2.14**

*Search terms found in the text are highlighted.*

Listing 2.16 is another code example that doesn't contain a Page\_Load event handler. Therefore, let's begin our examination with the HTML portion of the script (lines 16 through 31). Because Listing 2.16 uses post-back forms, a server-side form is created on line 18. Next, a pair of text boxes are created: the first, txtSearchTerms, will be the text box the user enters

his search term in (line 20); the second, `txtText`, is a multiline text box in which the user will enter the text to search (lines 23 and 24). On lines 26 and 27, a button control is created that, when clicked, will fire the `btnSubmit_OnClick` event handler. Finally, on line 28 a label control, `lblResults`, is created; this label control will display the user's text entered in `txtTerms` with all instances of the search term entered in `txtSearchTerms` highlighted.

The `btnSubmit_OnClick` function, starting at line 3, begins by reading in the value of the `txtSearchTerms` text box into a string variable, `strTerms` (line 5). Next, a `Regex` object instance is created with a pattern of the user-entered search term surrounded by `\b`. In regular expressions, `\b` is a special character representing a word boundary. Adding this both before and after the search term will have the effect of only highlighting search terms in the user-entered text that are their own words; that is, if the user enters a search term of "in" and the text, "I sleep in the incinerator," the word "in" will be highlighted, but the "in" in "incinerator" will not. Also, the "i" option is specified, indicating that the search and replace will be not case sensitive (line 6).

On lines 9 and 10, the `Replace` method of the `Regex` class is used. The `Replace` method accepts two parameters: the string to search for the pattern and the string to replace any found matches. In English, lines 9 and 10 say, "Search the contents of `txtText.Text` looking for any matches to the pattern (the search term as its own word). If you find any, replace it with a highlighted version of the search term."

The `Replace` method returns a string that has had all matches in its first parameter replaced by the second. On line 9 we set this return string equal to the variable `strNewText`. Finally, line 12 outputs the highlighted results, `strNewText`.

The `Regex` class contains a number of other useful methods. One really neat method that I encourage you to examine in detail on your own is the `Split` method. This method is similar to the `Split` method of the `String` class, except that instead of taking a string delimiter to split a string into an array, it accepts a regular expression as a delimiter!

## Generating Images Dynamically

There are many real-world scenarios in which the ability to create graphic images on-the-fly is needed. For example, imagine that you had a database table with monthly profits. It would be nice to be able to allow a user to visit a reporting page and have a chart dynamically created as a GIF file on the Web server and seamlessly embedded into the reports ASP.NET page.

This was an impossible task in classic ASP without the use of a third-party component (or without some very ugly and hacked together code). With ASP.NET, though, creating images on-the-fly in a wide variety of formats is quite possible and easy, thanks to the inherent support of image generation in the .NET Framework.

The .NET Framework contains an extensive drawing API, offering a multitude of classes with an array of methods and properties for drawing all sorts of shapes and figures. In fact, there is such an assortment of drawing functions that an entire book could be dedicated on the topic. Therefore, we will only look at a few of the more useful drawing functions. I highly suggest that you take ample time to root through all the classes in the `System.Drawing` and its derivative namespaces. There are many classes within these namespaces with literally hundreds of methods and properties!

When an image is dynamically created using the .NET Framework, it is created as an in-memory image. Methods can be used to send this in-memory image to disk on the Web server or to a stream (such as the Response stream). In this section we'll examine both how to save an image to file on the Web server and how to send a dynamically created image to the user's browser!

## Saving Dynamically Created Images on the Web Server

When creating images on-the-fly, the two most useful classes are the `Bitmap` and `Graphics` classes. The `Bitmap` class represents an instance of an image. The `Graphics` class contains the various methods for drawing lines, curves, ellipses, rectangles, and other geometric shapes.

Rather than presenting a lengthy tutorial on these two classes, let's look at a code example and then examine the methods and properties used to create a dynamic image. We will, of course, only be looking at a small subset of all the graphic functions. To learn more about the numerous graphic functions, refer to the `System.Drawing` namespace documentation in the .NET Framework Reference.

Listing 2.17 contains the code for a function that creates a very rudimentary bar chart and saves it as a file on the server. The function, `DrawBarGraph`, has the following definition:

```
DrawBarGraph(title, X_Data, Y_Data)
```

The first parameter, *title*, is of type `String` and is used to give a title to the chart. *X\_Data* and *Y\_Data* are both `ArrayLists`: *X\_Data* contains the X-axis labels for each data point, whereas *Y\_Data* contains each data point (and needs to be an integral value). Lines 57 through 72 in Listing 2.17 illustrate how to set up the data points and call the `DrawBarGraph` function. Although Listing 2.17 uses hard coded values for the X and Y axes, these values could have been easily pulled from a database. The output is shown in Figure 2.15.

---

### LISTING 2.17 With the .NET Framework You Can Create a Bar Chart On-the-Fly

```
1: <%@ Import Namespace="System.Drawing" %>
2: <%@ Import Namespace="System.Drawing.Imaging" %>
3: <script language="VB" runat="server">
```

**LISTING 2.17** Continued

```

4: Sub DrawBarGraph(strTitle as String, aX as ArrayList, aY as ArrayList)
5:     Const iColWidth as Integer = 60, iColSpace as Integer = 25, _
6:         iMaxHeight as Integer = 400, iHeightSpace as Integer = 25, _
7:         iXLegendSpace as Integer = 30, iTitleSpace as Integer = 50
8:     Dim iMaxWidth as Integer = (iColWidth + iColSpace) * aX.Count +
iColSpace, _
9:         iMaxColHeight as Integer = 0, _
10:        iTotHeight as Integer = iMaxHeight + iXLegendSpace + iTitleSpace
11:
12:    Dim objBitmap as Bitmap = new Bitmap(iMaxWidth, iTotHeight)
13:    Dim objGraphics as Graphics = Graphics.FromImage(objBitmap)
14:
15:    objGraphics.FillRectangle(new SolidBrush(Color.White), _
16:        0, 0, iMaxWidth, iTotHeight)
17:    objGraphics.FillRectangle(new SolidBrush(Color.Ivory), _
18:        0, 0, iMaxWidth, iMaxHeight)
19:
20:    ' find the maximum value
21:    Dim iValue as Integer
22:    For Each iValue in aY
23:        If iValue > iMaxColHeight then iMaxColHeight = iValue
24:    Next
25:
26:    Dim iBarX as Integer = iColSpace, iCurrentHeight as Integer
27:    Dim objBrush as SolidBrush = new SolidBrush(Color.FromARGB(70,20,20))
28:    Dim fontLegend as Font = new Font("Arial", 11), _
29:        fontValues as Font = new Font("Arial", 8), _
30:        fontTitle as Font = new Font("Arial", 24)
31:
32:    ' loop through and draw each bar
33:    Dim iLoop as Integer
34:    For iLoop = 0 to aX.Count - 1
35:        iCurrentHeight = ((aY(iLoop).ToDouble() / iMaxColHeight.ToDouble()) *
-
36:            (iMaxHeight - iHeightSpace).ToDouble())
37:
38:        objGraphics.FillRectangle(objBrush, iBarX, _
39:            iMaxHeight - iCurrentHeight, iColWidth, iCurrentHeight)
40:        objGraphics.DrawString(aX(iLoop), fontLegend, objBrush, iBarX,
iMaxHeight)
41:        objGraphics.DrawString(Int32.Format(aY(iLoop), "#,###"), fontValues,
-
42:            objBrush, iBarX, iMaxHeight - iCurrentHeight -
15)

```

**LISTING 2.17** Continued

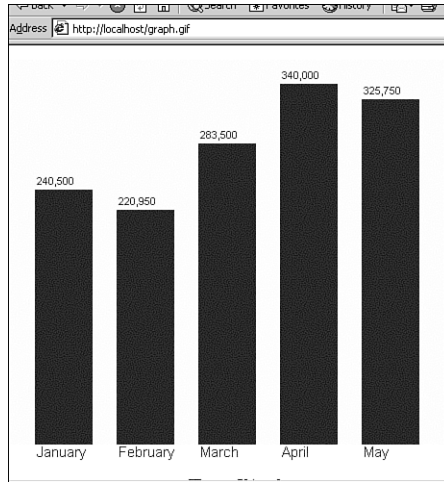
---

```
43:
44:     iBarX += (iColSpace + iColWidth)
45:     Next
46:
47:     objGraphics.DrawString(strTitle, fontTitle, objBrush, _
48:         (iMaxWidth / 2) - strTitle.Length * 6, iMaxHeight +
iXLegendSpace)
49:     objBitmap.Save("C:\inetpub\wwwroot\graph.gif", ImageFormat.GIF)
50:
51:     objGraphics.Dispose()
52:     objBitmap.Dispose()
53: End Sub
54:
55:
56: Sub Page_Load(source as Object, e as EventArgs)
57:     Dim aMonths as ArrayList = new ArrayList(), _
58:         aProfits as ArrayList = new ArrayList()
59:
60:     aMonths.Add("January")
61:     aMonths.Add("February")
62:     aMonths.Add("March")
63:     aMonths.Add("April")
64:     aMonths.Add("May")
65:
66:     aProfits.Add(240500)
67:     aProfits.Add(220950)
68:     aProfits.Add(283500)
69:     aProfits.Add(340000)
70:     aProfits.Add(325750)
71:
72:     DrawBarGraph("Profits!", aMonths, aProfits)
73: End Sub
74: </script>
```

---

Listing 2.17 begins with two `Import` directives that are useful when creating images on-the-fly. Line 1 imports the `System.Drawing` namespace, which contains most of the many classes we use in the code: `Bitmap`, `Graphics`, `Font`, `SolidBrush`, and so on; line 2 imports the `System.Drawing.Imaging` namespace, which contains the `ImageFormat` class that we use on line 49 to save the bar chart as a GIF image file.

The `DrawBarGraph` function begins on line 4, defined to accept three parameters: a string title, an `ArrayList` of labels for the X-axis, and an `ArrayList` of data points. Next, on lines 5 through 7, a number of useful constants are defined, and on lines 8 through 10, the variables we'll need for this function are created and assigned values.

**FIGURE 2.15**

*A bar chart with company profits has been dynamically created.*

On line 12, an instance of the `Bitmap` class is created. The version of the `Bitmap` constructor we've used in this example expects two integer values, a width and a height, which are defined by `iMaxWidth` and `iTotalHeight` in our example. Next, on line 13, an instance of the `Graphics` class is created. The `Graphics` class does not contain a constructor; rather, to get an instance of the `Graphics` class, we must use the static method `FromImage`. `FromImage` expects an `Image` object as its lone parameter and returns a `Graphics` object. (Because the `Bitmap` class is derived from the `Image` class, we can simply pass our `Bitmap` instance, `objBitmap`, to the `FromImage` method.)

On line 15, the entire image is painted white using the `FillRectangle` method of the `Graphics` class. Keep in mind that the `Graphics` class contains the various functions to draw lines, rectangles, ellipses, and other geometric shapes. All these functions require either a `Pen` or `Brush` instance, depending on whether a line or filled image is being drawn. On line 17 another rectangle is drawn with the `FillRectangle` method, this one serving as the backdrop for the graphed data. For more information on the `FillRectangle` method (or for information on the `Brush` and `SolidBrush` classes), be sure to refer to the .NET Framework Documentation.

Because we only have a fixed height to draw all the bars in the bar graph, we must first determine the largest value in our set of data points so that we can scale all the data points relative to the largest one. On lines 21 through 24, a simple `For Each ... Next` loop is used to iterate through the `ArrayList` of data points to find the largest value. This largest value is stored in `iMaxColHeight` (line 23).

Lines 26 through 30 define the variables we'll need for the actual graphing of the bar chart. Line 27 creates a dark red solid brush; lines 28 through 30 create three instances of the `Font` class, each of the Arial font family with varying point sizes. The font instances `fontLegend`, `fontValues`, and `fontTitle` are used to write the X-axis labels (represented by the values in `aX`), the data point values, and the title of the chart, respectively.

The code in lines 34 through 45 does the actual work of creating the graph. Line 34 starts a `For ... Next` loop to iterate through each element of the `aX` `ArrayList`. It is essential that the X-axis label `ArrayList` (`aX`) and the data points `ArrayList` (`aY`) contain the same number of elements. Furthermore, realize that `aX[N]` represents the X-axis label for data point `aY[N]`.

With each iteration of the loop, we will be drawing a particular bar in the bar chart. Lines 35 and 36 calculate the height for the current bar; this height is relative to tallest bar in the bar chart. A ratio is derived with `aY(iLoop).ToDouble() / iMaxColHeight.ToDouble()`, which is then multiplied with the highest bar chart value to obtain the scaled height for the current bar.

On lines 38 and 39, the actual bar is drawn with the `FillRectangle` method. Line 40 draws the X-axis label at the bottom of the bar, whereas lines 41 and 42 draw the data point value at the top of the bar. This function assumes that the values of the data points are going to be numeric and, based on that assumption, formats the value using the `Int32.Format` method (line 41). Finally, line 44 increments the current x position, `iBarX`.

By the time the code execution reaches line 47, all the bars have been drawn and labeled. Line 47 simply adds the bar chart title to the image. Line 49 uses the `Save` method of the `Image` class (which the `Bitmap` class inherits). In Listing 2.17, the `Save` method expects two parameters: the first parameter it expects needs to be a string that represents the full path of the file to save to image to; the second parameter must be an instance of the `ImageFormat` class. The `ImageFormat` class defines the low-level details on how an image is to be physically stored. A number of static properties of the `ImageFormat` class return `ImageFormat` instances. A listing of these image formats can be seen in Table 2.3.

**TABLE 2.3** The `ImageFormat` Class Contains a Number of Properties Representing Various Image Formats

<i>Property</i>	<i>Description</i>
BMP	Specifies a bitmap image.
EMF	Specifies the Windows enhanced metafile image format.
EXIF	Specifies the Exchangeable Image Format.
FlashPIX	Specifies the FlashPIX image format.
GIF	Specifies the GIF image format.
Icon	Specifies a Windows icon image format.



**TABLE 2.3** Continued

<i>Property</i>	<i>Description</i>
JPEG	Specifies the JPEG image format.
MemoryBMP	Specifies the memory bitmap image format.
PhotoCD	Specifies the Eastman-Kodak PhotoCD image format.
PNG	Specifies the PNG image format.
TIFF	Specifies the TIFF image format.
WMF	Specifies the Windows metafile image format.

After line 49 has executed, the bar chart has been saved as a file to the Web server, indicated by the path specified as the first parameter to the Save method. To conclude our DrawBarChart function, we clean up the resources used by calling the Dispose method for the Graphics class and Bitmap class instances.

In Listing 2.17, the DrawBarChart function is called from the Page\_Load event handler. Before calling the function, however, two ArrayLists must be built and populated with the X-axis labels and data points. These two ArrayLists are created in Listing 2.17 on lines 57 and 58. The X-axis labels are populated in lines 60 through 64; the data points are assigned in lines 66 through 70. On line 72, the DrawBarChart function is called and passed the two ArrayLists along with the chart's title, "Profits!"

## Sending Dynamically Created Images to the Browser

In the previous section, we examined how to create and save a dynamic image to the Web server's file system through an ASP.NET page. Although this capability is quite useful, at times it would be nice to bypass this step and stream the dynamically created image directly to the Web visitor's browser.

Fortunately, this is possible with the Save method of the Image class. In Listing 2.4.1, we looked at one use of the Save method, passing it a file path on the Web server and an ImageFormat. The Save method can also accept a Stream object as its first parameter; in doing so, the Save method will send the output of the image to the stream as opposed to disk.

With ASP.NET, the Response object has a number of new properties and methods. One such property is the OutputStream property, which provides programmatic access to the binary data being sent to the client's browser. The Save method of the Image class can use the Response object's OutputStream property to send a dynamically created, in-memory image directly to the client! Listing 2.18 contains a short code example in which a standard advertising banner is created on-the-fly and squirted to the user's browser.

**LISTING 2.18** Dynamically Created Images Can Be Sent Directly to the User's Browser

---

```
1: <%@ Import Namespace="System.Drawing" %>
2: <%@ Import Namespace="System.Drawing.Imaging" %>
3: <script language="VB" runat="server">
4: Sub Page_Load(source as Object, e as EventArgs)
5:   Dim objBitmap as Bitmap = new Bitmap(468, 60)
6:   Dim objGraphics as Graphics = Graphics.FromImage(objBitmap)
7:   Dim objBrush as SolidBrush = new SolidBrush(Color.FromARGB(0,80,80)), _
8:     objBlackBrush as SolidBrush = new SolidBrush(Color.Black), _
9:     objYellowBrush as SolidBrush = new SolidBrush(Color.Yellow)
10:
11:   Dim fontTitle as Font = new Font("Arial", 24), _
12:     fontSubtitle as Font = new Font("Arial Black", 9)
13:
14:   objGraphics.FillRectangle(new SolidBrush(Color.Ivory), 0, 0, 468, 60)
15:   objGraphics.DrawString("When you think ASP, think...", _
16:     fontSubtitle, objBlackBrush, 5, 8)
17:   objGraphics.DrawString("4GuysFromRolla.com", fontTitle, objBrush, 10, 20)
18:
19:   ' Draw a smiley face.. first draw the yellow circle!
20:   objGraphics.FillEllipse(objYellowBrush, 375, 5, 50, 50)
21:
22:   ' Now create the eyes.
23:   objGraphics.FillEllipse(objBlackBrush, 387, 20, 6, 6)
24:   objGraphics.FillEllipse(objBlackBrush, 407, 20, 6, 6)
25:
26:   ' And finally the smile.
27:   Dim aPoints(4) as Point
28:   aPoints(0).X = 383 : aPoints(0).Y = 35
29:   aPoints(1).X = 395 : aPoints(1).Y = 45
30:   aPoints(2).X = 405 : aPoints(2).Y = 45
31:   aPoints(3).X = 417 : aPoints(3).Y = 35
32:   objGraphics.DrawCurve(new Pen(Color.Black), aPoints)
33:
34:   Response.ContentType = "image/jpeg"
35:   objBitmap.Save(Response.OutputStream, ImageFormat.JPEG)
36:
37:   objGraphics.Dispose()
38:   objBitmap.Dispose()
39: End Sub
40: </script>
```

---

Listing 2.18 begins as Listing 2.17 did: by Importing the System.Drawing and System.Drawing.Imaging namespaces. Next, in the Page\_Load event handler, a 468×60

Bitmap instance is created (the dimensions of the standard advertising banner; on line 5). On line 6, a Graphics instance is created and instantiated with the Graphics.FromImage method. On lines 7 through 9, a number of SolidBrushes are created and instantiated; lines 11 and 12 create the two Fonts we'll be using.

From lines 14 through 32, a number of Graphics methods are called to create the image. Finally, after the image is created, it's time to send it to the browser. Before we send the actual binary content to the browser, however, we set the binary data's ContentType as "image/jpeg" (line 34). This ContentType informs the browser on how to display the binary information that it is receiving.

To actually send the binary data of the image, we use the Save method. Note that, on line 35, instead of specifying a filename as the first parameter, we specify the Stream to write the image to. Because we want to send the image to the browser, we specify that the OutputStream of the Response object be used. Also, the image should be converted and sent as a JPEG. Listing 2.18 concludes with a little house cleaning on lines 37 and 38.

One way to view the dynamically created image in Listing 2.18 is to visit the ASP.NET page that creates it and squirts it out to the OutputStream of the Response object. Figure 2.16 depicts a browser visiting the ASP.NET page directly.



**FIGURE 2.16**

*The dynamically created image can be viewed by directly visiting the ASP.NET page.*

Ideally, it would be nice to be able to display both content and the dynamically generated graphic on the same page. This, however, cannot be done through the same ASP.NET page; for example, if you were to add `Response.Write("Hello, World!")` in line 33 of Listing 2.18, instead of seeing an image when visiting the ASP.NET page, you'd see a broken image link. Essentially, this is because the browser is being sent "Hello, World!" and then the binary data for the image, and it assumes that the "Hello, World" is part of the binary data for the image.

If you want to display both textual content and a dynamically created image, you must create a standard HTML or ASP.NET page and then refer to the dynamic image content using HTML IMG tags. Listing 2.19 contains a very short example of an ASP.NET page that displays HTML content and the dynamic image created in Listing 2.18. Listing2.18.aspx, the ASP.NET page that creates the image, is referred to in the IMG tag as if it were a JPG image itself. Figure 2.17 contains a screenshot of Listing 2.19 when viewed through a browser.

**LISTING 2.19** To View a Dynamically Created Image from an HTML Page, Use the IMG tag

```
1: <html>
2: <body>
3:   <h1>Hello, World!</h1>
4:   Here is a picture of my banner!
5:   <p>
6:     
8:   </p>
9: </body>
10: </html>
```

**FIGURE 2.17**

The output of Listing 2.19, when viewed through a browser

**TIP**

When viewing a dynamically created image from an ASP.NET page through an IMG tag, you can pass parameters through the QueryString such as: ``. The ASP.NET page can then read the QueryString value and alter its actions based on such information!

## Sending E-mail from an ASP.NET Page

With classic ASP, there were a variety of ways to send an e-mail from an ASP page. Microsoft included its own free component, *CDONTS (Collaborative Data Object for NT Server)*, but there were also a number of commercially (and freely) available third-party e-mailing components.

The .NET Framework contains two classes to assist with sending e-mail. The first class, `MailMessage`, defines the various properties of the e-mail message itself, such as the e-mail message body, the sender, the receiver, the subject, the priority, and so on. The second class, `SmtpMail`, performs the work of actually sending the e-mail message. To send e-mails through

an ASP.NET page, you will need to have the SMTP Service installed on your Web server. This is a standard installation option when installing IIS on Windows NT 4.0 or Windows 2000.

Listing 2.20 depicts a very simple product information request form. The user can enter his name and e-mail address to receive further product information by e-mail. Upon submission of the form, an e-mail is sent to the user providing information on the various product lines.

### LISTING 2.20 The .NET Framework Provides Inherent E-mail-Sending Support

```

1: <%@ Import Namespace="System.Web.Util" %>
2: <script language="VB" runat="server">
3: Sub btnSubmit_OnClick(source as Object, e as EventArgs)
4:   'Send an email address with product information
5:   Dim objMessage as New MailMessage
6:   objMessage.BodyFormat = MailFormat.Html
7:   objMessage.To = txtEmail.Text & " (" & txtName.Text & ")"
8:   objMessage.From = "webmaster@acme.com (Product Information)"
9:   objMessage.Headers.Add("Reply-To", "questions@acme.com")
10:  objMessage.Priority = MailPriority.High
11:  objMessage.Subject = "Product Information Request"
12:
13:  objMessage.Body = "<i>Hello " & txtName.Text & "</i>! To learn more
about our " & _
14:                    "various products, visit the following URLs:<p><ul>" & _
15:                    "<li><a
href=""http://www.acme.com/Widgets.htm"">Widgets</a>" & _
16:                    "<li><a
href=""http://www.acme.com/DooHickies.htm"">DooHickies</a>" & _
17:                    "<li><a href=""http://www.acme.com/TAMB.htm"">Thning-a-ma-
Bobs</a>" & _
18:                    "</ul><p> If you have any " & _
19:                    "questions, please simply reply to this email!<p><hr><p>" &
_
20:                    "<font color=red size=-1><i>Thank you for choosing
Acme!</i></font>"
21:
22:
23:   'Send the message
24:   Smtplib.Send(objMessage)
25:
26:   'Display notification of mail being sent
27:   Response.Write("<font color=red><b>You have been sent " & _
28:                  "product information! Thank you!</b></font>")
29: End Sub
30: </script>
31:

```

**LISTING 2.20** Continued

```
32: <html>
33: <body>
34:   <form method=post runat=server>
35:     <h1>Information Request</h1>
36:     <b>Your Name:</b>
37:     <asp:textbox id="txtName" runat="server" /><br>
38:     <b>Your Email Address:</b>
39:     <asp:textbox id="txtEmail" runat="server" /><p>
40:
41:     <asp:button id="btnSubmit" runat="server" OnClick="btnSubmit_OnClick"
42:               text="Send me Product Information" />
43:   </form>
44: </body>
45: </html>
```

Listing 2.20 starts by importing the namespace that the `MailMessage` and `SmtpMail` classes exist under: `System.Web.Util`. Because there is no `Page_Load` event handler, let's next move on to examining the HTML content from line 32 through line 45. Note that a post-back, server-side form control is used (line 34). Next, two text boxes are created: the first, `txtName`, is for the user's name (line 37), whereas the second, `txtEmail` is for the user's e-mail address (line 39). Finally, a button control is used on lines 41 and 42. When clicked, the form will be posted back and the `btnSubmit_OnClick` event handler will fire.

The `btnSubmit_OnClick` event handler (lines 3 through 29), sends an e-mail to the e-mail address specified by the user in the `txtEmail` text box. As mentioned earlier, sending e-mails involves two classes, `MailMessage` and `SmtpMail`. On line 5 a new instance of the `MailMessage` class is created and, in lines 6 through 20, some of its many properties are set.

Some of the properties for the `MailMessage` include a `BodyFormat` property (line 6), which can be set to either `MailFormat.Text` or `MailFormat.Html`. On line 7 and 8, we set the `To` and `From` e-mail addresses using the following format:

```
Email-address (Display Name)
```

Many e-mail clients will pick out the text between the parenthesis and display that in the `From` of the e-mail message as opposed to the actual e-mail address itself. On line 9 we access the `Headers` property of the `MailMessage` class. This property is an `IDictionary` interface, meaning that it can be treated similar to a `Hashtable`. On line 9 we add the `Reply-To` header to the outgoing e-mail address so that if the user replies to the product information-request e-mail, it will be sent to `questions@acme.com` as opposed to `webmaster@acme.com`. Next, the `Priority` of the message is set. This property can be set to one of three values: `MailPriority.Low`, `MailPriority.Normal` (the default), or `MailPriority.High`. Line 11 sets the e-mail's subject, whereas line 13 defines the HTML-formatted body for the e-mail message.

If you've worked with the CDONTS component with classic ASP, you'll no doubt feel at home with the properties of the `MailMessage` class. The `MailMessage` and CDONTS component are almost syntactically identical. One major difference, however, is that the `MailMessage` class does not encompass any functionality for sending a message. That, instead, is left up to the `SmtpMail` class.

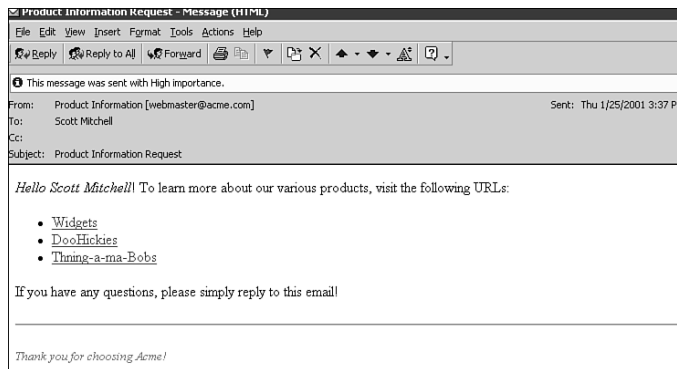
The `SmtpMail` class is a very bare-bones class. It's sole purpose is to send a `MailMessage` instance using the SMTP protocol. It contains a single shared function, `Send`, which expects a `MailMessage` object instance. Line 24 illustrates the use of the `Send` method.

Finally, on lines 27 and 28, a message is displayed indicating to the user that he can expect to receive an e-mail with product information. Figure 2.18 contains a screenshot of Listing 2.20 after the user has entered his name and e-mail address. Figure 2.19 contains a screenshot of the e-mail received by the user. Note that a high priority is set; the e-mail is displayed as coming from Product Information as opposed to just `webmaster@acme.com`; the e-mail is addressed to the user's name, not to his e-mail address; and the e-mail is HTML-formatted.

2

COMMON ASP .NET  
CODE TECHNIQUES**FIGURE 2.18**

The output of Listing 2.20, when viewed through a browser.

**FIGURE 2.19**

An HTML-formatted, high priority e-mail is sent to the user with information on Acme products.

## Network Access Via an ASP.NET Page

With classic ASP, performing network access through an ASP page was impossible without the use of a third-party or custom-developed COM component. For example, grabbing the HTML output of a Web page on a remote Web server via an ASP page was only possible with a component of some kind, such as ASPHTTP from ServerObjects.com. The .NET Framework, however, contains a plethora of classes to assist with network access. These numerous classes are all located in the `System.Net` namespace.

A very common need for Web developers is the ability to grab the HTML content of a Web page on a remote server. Perhaps the developer wants to perform a *screen scrape*, grabbing specific portions of the HTML output to integrate into his own Web page. To assist in this task, the .NET Framework provides two classes—`WebResponse` and `WebRequest`—that can be used for accessing information over the Internet.

`WebRequest` and `WebResponse` are both abstract classes, meaning that you cannot directly create an instance of the class using the `new` keyword. Rather, as we will examine shortly, instances for the `WebRequest` and `WebResponse` classes must be created via the `WebRequestFactory` class and the `GetResponse()` method of the `WebRequest` class, respectively.

Listing 2.21 contains a “Poor Man’s Internet Explorer,” a browser within a browser. The code in Listing 2.21, when viewed through a browser, presents the user with a text box in to which he can enter a fully qualified URL (such as `http://www.4GuysFromRolla.com`). After the user enters a URL and clicks the Go button, the Web page he entered is displayed along with the Response and Request headers. Output is shown in Figure 2.20.

---

### LISTING 2.21 The .NET Framework Provides Internet Access from an ASP.NET Page

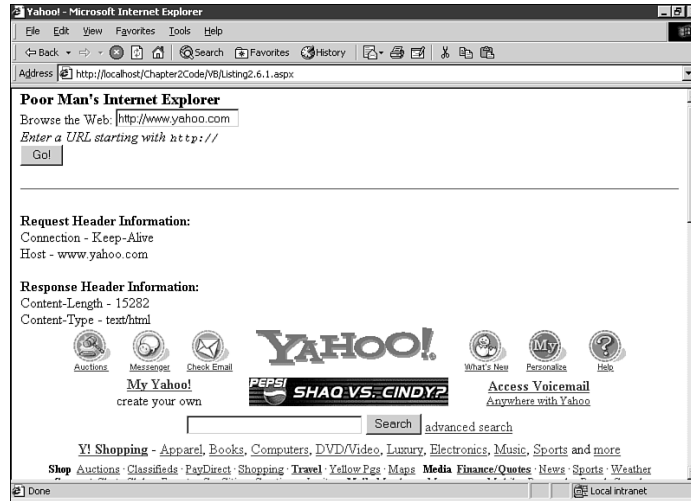
---

```
1: <%@ Import Namespace="System.IO" %>
2: <%@ Import Namespace="System.Net" %>
3: <script language="C#" runat="server">
4:     void btnSubmit_OnClick(Object source, EventArgs e)
5:     {
6:         // Create a WebRequest
7:         WebRequest wrRequest;
8:         wrRequest = WebRequestFactory.Create(txtURL.Text);
9:
10:        // Get the Response from the Request
11:        WebResponse wrResponse = wrRequest.GetResponse();
12:
```



**LISTING 2.21** Continued

```
13: // Display the Request headers
14: lblHTML.Text = "<b>Request Header Information:</b><br>";
15: foreach (String strHeader in wrRequest.Headers)
16:     lblHTML.Text += strHeader + " - " +
17:         wrRequest.Headers[strHeader] + "<br>";
18:
19: // Display the Response headers
20: lblHTML.Text += "<p><b>Response Header Information:</b><br>";
21: foreach (String strHeader in wrResponse.Headers)
22:     lblHTML.Text += strHeader + " - " +
23:         wrResponse.Headers[strHeader] + "<br>";
24:
25: // Read the Response into a stream and output the stream
26: Stream objStream = wrResponse.GetResponseStream();
27: StreamReader objStreamReader = new StreamReader(objStream);
28:
29: lblHTML.Text += objStreamReader.ReadToEnd();
30: }
31: </script>
32:
33: <html>
34: <body>
35:     <form runat="server">
36:         <font size=+1><b>Poor Man's Internet Explorer</b></font>
37:         <br>Browse the Web:
38:         <asp:textbox id="txtURL" runat="server" /><br>
39:         <i>Enter a URL starting with <code>http://</code></i><br>
40:         <asp:button id="btnSubmit" runat="server" Text=" Go! "
41:             OnClick="btnSubmit_OnClick" />
42:
43:         <p><hr><p>
44:         <asp:label id="lblHTML" runat="server" />
45:     </form>
46: </body>
47: </html>
```

**FIGURE 2.20**

*This ASP.NET page acts like an extremely simple browser.*

Listing 2.21 starts out with some familiar lines of code. Line 2 Imports the `System.Net` namespace, which contains the `WebResponse`, `WebRequest`, and `WebRequestFactory` classes that we'll be using. The `System.IO` namespace is also Imported because we will be using the `Stream` and `StreamReader` classes to access the response HTML. (We discussed the `Stream` and `StreamReader` classes earlier in the section "Working with the File System.")

Because Listing 2.21 contains no `Page_Load` event handler, let's start our examination of the code with the HTML content. On line 35 a post-back form is created. Inside this form are two form elements: a text box, `txtURL`, into which the user will enter the URL she wants to visit (line 38); and a Go button, `btnSubmit`, for the user to click to load the entered URL (lines 40 and 41). Note that the button has its `OnClick` event wired up to the `btnSubmit_OnClick` event handler. When this button is clicked, the form will be submitted; and, when the page reloads, the `btnSubmit_OnClick` event handler will execute.

The `btnSubmit_OnClick` event handler, starting on line 4, is responsible for grabbing the HTML from the URL entered by the user in the `txtURL` text box. To grab the HTML at the URL specified by the user, we must go through a two-step process:

1. We need to make an HTTP request to the URL using the `WebRequest` class. On line 7 a variable of type `WebRequest` is created; on line 8, this variable is instantiated, assigned to the `WebRequest` instance returned by the `Create` method of the `WebRequestFactory`.
2. We need to grab the response stream from the Web request we just made; the `WebResponse` class handles this. On line 11 a `WebResponse` variable is created and assigned to the `WebResponse` object returned by the `GetResponse()` method of the `WebRequest` class.

Whenever a client sends an HTTP request to the server, a number of optional headers are usually passed along with various bits of information. Likewise, when the server returns the content to the client, a number of headers can be sent along with the data. The `WebRequest` and `WebResponse` classes both have a `Headers` property, which represents an instance of the `WebHeaders` class. Through this property, a developer can programmatically send request headers and iterate through the response headers.

To explicitly send a header to the server when making a Web request from an ASP.NET page, simply use the following syntax *prior* to calling the `GetResponse()` method:

```
WebResponseInstance.Headers.Add(HeaderName, HeaderValue);
```

Therefore, if you wanted to add the specific request header `Foo: Bar`, in Listing 2.21 you would need to add the following line of code on line 9:

```
wrRequest.Headers.Add("Foo", "Bar");
```

Recall that the `Header` property returns a `WebHeaders` class instance. The `WebHeaders` class is derived from the `NameValueCollection` class, which means that we can treat the `Headers` property similar to an ordinary `Hashtable` collection. On lines 14 through 17, the request headers are displayed using a `foreach` loop. On lines 20 through 23, the response headers are displayed in a similar fashion.

Next, we need to pick out the content from the `WebResponse` variable `wrResponse`. The `WebResponse` class contains a `GetResponseStream()` method, which returns a `Stream` object that contains the HTML returned by the URL specified by the `WebRequest` instance. On line 26, a `Stream` instance is created, `objStream`, and is assigned to the `Stream` instance returned by the `GetResponseStream()` method. To read the contents of this `Stream`, we must use the `StreamReader` class. On line 27, a `StreamReader` class is instantiated, passing the `objStream` variable to the constructor. Finally, the complete contents of the `Stream` are displayed on line 29 via the `ReadToEnd()` method of the `StreamReader` object.

## NOTE

Because the `WebRequest` method is abstract, you cannot directly create an instance of the class using the `new` keyword; rather, you must use the `Create` method of the `WebRequestFactory` (see line 8 in Listing 2.21). Similarly, the `WebResponse` class is also abstract. On line 11, an instance of this class is obtained via the `GetResponse()` method of the `WebRequest` class.

Another useful class in the `System.Net` namespace is the `DNS` class. (This capability was not possible in classic ASP without the use of a third-party component.) This class can be used to

resolve DNS hostnames into IP addresses. Such a technique could be used to verify e-mail addresses by ensuring that the domain name specified by the user actually resolved to an existing domain name.

Listing 2.22 illustrates how to use the DNS classes `Resolve` method, along with a regular expression, to build a fairly reliable e-mail address validation ASP.NET page. Of course the only way to *truly guarantee* that a user has entered his own valid e-mail address is to require him to respond to a confirmation e-mail message. However, the technique shown in Listing 2.22 is more reliable than *just* checking the e-mail address against a regular expression.

### LISTING 2.22 ASP.NET Provides Built-in DNS Resolving Support

```
1: <%@ Import Namespace="System.Net" %>
2: <%@ Import Namespace="System.Net.Sockets" %>
3: <%@ Import Namespace="System.Text.RegularExpressions" %>
4: <script language="VB" runat="server">
5:     Sub btnSubmit_OnClick(source as Object, e as EventArgs)
6:         'Check to make sure that the email addy is in the right form
7:         Dim strEmail as String, strPattern as String
8:         strEmail = txtEmail.Text
9:         strPattern = "^[\\w-_.]+@[([\\w]+\\.)+\\w+$"
10:
11:         If Not Regex.IsMatch(strEmail, strPattern, "i") then
12:             'Invalid email address form!
13:             Response.Write("<font color=red><i>Your email address is in an " & _
14:                 " illegal format.</i></font><p>")
15:         Else
16:             'Check to see if the domain name entered in the email address exists
17:             Dim strDomain as String
18:             strDomain = strEmail.Substring(strEmail.IndexOf("@") + 1)
19:
20:             'Attempt to Resolve the hostname
21:             Dim strIP as String
22:             try
23:                 strIP = DNS.Resolve(strDomain).ToString()
24:
25:                 'If we reach here, we have a valid email address, so do whatever
26:                 'processing or whatnot needs to be done...
27:                 Response.Write("<b>Valid email address. Your domain name has " & _
28:                     "an IP of " & strIP & ". Thank you!</b>")
29:             catch se as SocketException
30:                 'The DNS resolve was unsuccessful...
31:                 strIP = se.Message
32:
```

**LISTING 2.22** Continued

```
33:         Response.Write("<font color=red><i>" & strIP & "</i></font><p>")
35:     end try
36: End If
37: End Sub
38: </script>
39:
40: <html>
41: <body>
42:     <form runat="server">
43:         <br><b>Enter your Email address:</b>
44:         <asp:textbox id="txtEmail" runat="server" />
45:         <p>
46:         <asp:button id="btnSubmit" runat="server" Text="Check Email"
47:             OnClick="btnSubmit_OnClick" />
48:     </form>
49: </body>
50: </html>
```

Listing 2.22 begins by importing three namespaces: `System.Net`, which contains the definition for the `DNS` class; `System.Net.Sockets`, which contains the definition for the `SocketException` class that we'll need to use if the user enters an e-mail address with an invalid domain name; and `System.Text.RegularExpressions` because we are going to use a regular expression to ensure that the e-mail address is in the proper format.

The code in Listing 2.22 creates a post-back form with a text box, `txtEmail`, for the user to enter his e-mail address (lines 42 and 44, respectively). The page also displays a button titled `Check Email`, which, when clicked, will submit the form, causing the `btnSubmit_OnClick` event handler to fire. This button, `btnSubmit`, is created on lines 46 and 47.

The task of the `btnSubmit_OnClick` event handler is to ensure that the user has entered a valid e-mail address. We begin by reading the value of the `txtEmail` text box in to a `String` variable, `strEmail` (line 8). Then, on line 9, we create a regular expression pattern, `strPattern`, which, in short, looks for one or more characters preceding the `@` sign, followed by one or more of a number of characters followed by a period, and then followed by another grouping of one or more characters. Whew, that sounds overly complex! Basically we are trying to ensure that the user has entered the following:

```
SomeCharacters@(SomeCharacters.(ONE OR MORE TIMES))SomeCharacters
```

Next, on line 11, we call the static version of the `IsMatch` method of the `Regex` class to determine if our pattern is found in the e-mail address. If it is *not*, lines 12 through 14 are executed, which display an error message. If the pattern is found, the code following the `Else` statement on line 15 is executed.

**NOTE**

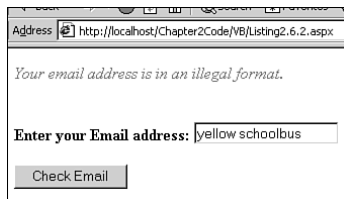
We could have simply used a `RegularExpressionValidator` control in our HTML document to ensure that the e-mail address was valid. For more information on the `RegularExpressionValidator` control, refer to Chapter 3, “Form Field Input Validation.”

If the e-mail address is in the proper format, all that is left to do is ensure that the domain name portion of the e-mail address is valid. To do this, we must first snip out the domain name from the e-mail address, which we do on line 18 using the `Substring` and `IndexOf` methods of the `String` class. When we have the domain name, we are ready to try to resolve it in to an IP address—if we can resolve the domain name in to an IP address, the domain name is valid; otherwise it is invalid.

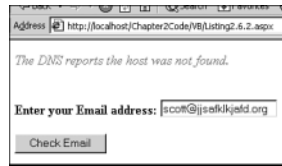
The `Resolve` method of the `DNS` class resolves a domain name (in the form of a string) into an instance of the `IPAddress` class. If the domain name cannot be resolved into an IP address, a `SocketException` is thrown. For that reason, we must have the call to the `Resolve` method in a `Try ... Catch` block. On line 22 we begin the `Try ... Catch` block and, on line 23, make a call to `Resolve`, a static method. Because the `Resolve` method returns an `IPAddress` instance, we must call the `ToString()` method of the `IPAddress` class before assigning it to our string variable, `strIP`.

If the domain name is successfully resolved in to an IP address, line 27 will be reached, where a message is displayed informing the user that his e-mail address is valid (and also displaying the resolved IP address). If, however, the domain name was not resolved to an IP address, the `Catch` portion of the `Try ... Catch` block will be reached (line 29). On line 31, `strIP` is assigned to the value of the `Message` property of the `SocketException` exception. On line 33, this `Message` is displayed as an error message to the user.

Figures 2.21, 2.22, and 2.23 show the output of Listing 2.22 for various scenarios. Figure 2.21 shows the error message a user will receive if she enters her e-mail address is an invalid format; Figure 2.22 depicts the output the user will be presented with if he enters an invalid domain name in his e-mail address; and, finally, Figure 2.23 shows the output of Listing 2.22 when the user enters an e-mail address in a valid format with a valid domain name.

**FIGURE 2.21**

*The user has entered an invalid e-mail address.*

**FIGURE 2.22**

*This user has entered an invalid domain name in her e-mail address.*

**FIGURE 2.23**

*The user has entered a valid e-mail address.*

Understand that a clever user could easily bypass this check system by entering an invalid username portion of his e-mail address with a valid domain name. For example, the domain name yahoo.com is, of course, valid. However, I suspect the username `ThisIsAReallyLongNameAndIAMTryingToProveAPoint` is registered. Therefore, if a user were to enter the following:

```
ThisIsAReallyLongNameAndIAMTryingToProveAPoint@yahoo.com
```

The script would identify that e-mail address as valid, even though it clearly does not really exist. As aforementioned, if you must guarantee that a user enters a valid e-mail address, it is imperative that you send him a confirmation e-mail and require him to respond.

## Uploading Files from the Browser to the Web Server Via an ASP.NET Page

Whether a user on Monster.com needs to upload her resume, or a GeoCities member wants to upload his personal Web site, there are a vast number of practical scenarios in which the ability for a visitor to upload a file (or files) from his computer to the Web site is essential. With classic ASP, this task usually fell on the shoulders of a third-party component, such as ASPUpload or SA-FileUp. With ASP.NET, however, file upload capabilities are available without the need for any third-party component.

When uploading a file from an HTML form, there are a few guidelines you must follow. First, and most importantly, you must specify the `multipart/form-data` encoding type in the form tag. This encoding type is specified in the form tag in the following syntax:

```
<form ... enctype="multipart/form-data" ...>
```

Next, you must provide an `INPUT` form element that has its `TYPE` property set to `FILE`. These types of `INPUT` form elements are displayed in a user's browser as a text box that contains a Browse button next to it. If the user clicks the Browse button, she will be able to select a file from her hard drive. When she does, the filename will appear in the associated text box. The following HTML would generate such a file upload text box with an accompanying Browse button:

```
<form enctype="multipart/form-data" method="post">
  <input type="file" name="fupUpload">
</form>
```

Because the previous snippet of code is simply HTML, you could easily add such code to a classic ASP page. The problem was actually retrieving the contents of the uploaded file and saving it to the Web server. Although this could be done with script alone, the implementation was usually difficult and messy enough to warrant the use of a third-party component. (For information on a script-only upload scenario with classic ASP, be sure to read <http://www.asp101.com/articles/jacob/scriptupload.asp>.)

#### NOTE

The complete documentation for uploading files through an HTML form can be found in RFC 1867, "Form-based File Upload in HTML," at <http://www.ietf.org/rfc/rfc1867.txt>.

With ASP.NET, an entire class exists to handle the uploaded file and its contents. This class, `HttpPostedFile`, enables a developer to either programmatically save the uploaded file on the Web server's file system or to access the contents of the uploaded file via a `Stream`. In this section we'll look at two examples: Listing 2.23 demonstrates how to provide a form to allow the user to upload a file and have it saved on the Web server; Listing 2.24 illustrates an ASP.NET page that accepts an uploaded image and reports the image's properties. When examining this listing, note that the image is never saved to the Web server's file system.

## Saving an Uploaded File to the Web Server's File System

When viewed through a browser, Listing 2.23 presents the user with a text box and Browse button, enabling the user to select a file from her hard drive that she wanted to upload to the Web server. If the user attempts to upload a file that does not exist, a warning message will be displayed. If, however, the user enters a valid filename (either by typing it in the text box or selecting the file via the Browse button) and the file is successfully saved on the Web server, a



confirmation message will be displayed, and the user will be presented with a link to view the uploaded content through the Web server. (Refer to Figure 2.24 to see a screenshot of the user's browser after successfully uploading a file.)

### LISTING 2.23 Component-less File Upload Is Possible via an ASP.NET Page

```
1: <%@ Import Namespace="System.IO" %>
2: <script language="VB" runat="server">
3:   Sub Page_Load(source as Object, e as EventArgs)
4:     'Has the form been submitted?
5:     If Page.IsPostBack AND fupUpload.PostedFile <> Nothing Then
6:       'Save the file if it has a filename and exists...
7:       If fupUpload.PostedFile.FileName.Trim().Length > 0 AND _
8:         fupUpload.PostedFile.ContentLength > 0 then
9:
10:        Const strBaseDir as String = "C:\My Projects\Uploaded Files\"
11:        Dim strFileName as String = _
12:          File.GetFileNameFromPath(fupUpload.PostedFile.FileName)
13:
14:        fupUpload.PostedFile.SaveAs(strBaseDir & strFileName)
15:
16:        'File has been saved!
17:        lblResults.Text = "<hr><p>Your file, " &
fupUpload.PostedFile.FileName & _
18:          ", has been saved to the Web server!<br>" & _
19:          "[<a href=""/Upload/" & strFileName & "">View
File</a>]"
20:      End If
21:    Else
22:      lblResults.Text = "<hr><p>Enter a filename to upload!"
23:    End If
24:  End Sub
25: </script>
26:
27: <html>
28: <body>
29:   <form runat="server" EncType="multipart/form-data">
30:     <h1>File Upload</h1>
31:     <b>Select the file to upload:</b><br>
32:     <input runat="server" id="fupUpload" type="file" >
33:     <p>
34:     <asp:button id="btnSubmit" runat="server" Text="Upload File" />
35:     <p><asp:label runat="server" id="lblResults" />
36:   </form>
37: </body>
38: </html>
```

**FIGURE 2.24**

*A file has been uploaded from the user's hard drive to the Web server's hard drive.*

Listing 2.23 begins by importing the `System.IO` namespace. This namespace is imported because, on line 12, the `GetFileNameFromPath` method of the `File` class is used to snip out the filename of the uploaded file. No namespaces need to be imported to use those classes that assist with browser-based file uploads.

Although we have a `Page_Load` event handler in this ASP.NET page, the majority of the code in the event handler only runs if the form has been posted back. For that reason, let's first examine the HTML content from lines 27 through 38. This content begins with form tag that has both the `runat="server"` attribute set as well as the `enctype="multipart/form-data"` setting (line 29). It is vitally important that this `enctype` parameter be included in the form tag, else the file upload will not work.

Next, on line 32, a file upload HTML control is created with the `id` `fupUpload`. Note that its `type` property is set to `file`; again, it is essential that this `INPUT` tag have its `type` property set to `file` for the file upload to work properly. It is important to set the `runat="server"` property of the `INPUT` tag so that you can programmatically access the uploaded file. On line 34 a button control, `btnSubmit`, exists. When this is clicked, the form will be submitted. Finally, on line 35, a label control, `lblResults`, is created, which will display a success message after the file is uploaded from the client's machine to the Web server.

The `Page_Load` event handler begins on line 3 and immediately checks for two things: to see if the page has been posted back to, and to ensure that the `PostedFile` property of the `fupUpload` file upload control is assigned a value. `PostedFile` is a property of the `HtmlInputFile` class and represents the file uploaded from the client to the Web server.

This `PostedFile` property is an instance of the `HttpPostedFile` class. The `HttpPostedFile` class has four very useful properties: `ContentLength`, which represents the size, in bytes, of the uploaded file; `ContentType`, which represents the MIME content type of the uploaded file; `FileName`, which represents the full path and filename on the client's computer of the uploaded file; and `InputStream`, which provides a `Stream` instance for access to the uploaded file's bits.

- `ContentLength`—Represents the size, in bytes, of the uploaded file.
- `ContentType`—Represents the MIME content type of the uploaded file.
- `FileName`—Represents the full path and filename on the client's computer of the uploaded file.
- `InputStream`—Provides a `Stream` instance for access to the uploaded file's bits.

If the form has been posted back and the `PostedFile` property is not equal to `Nothing`, the code between lines 6 and 20 will execute. If either of these conditions is `False`, line 22 will execute, displaying a message to the user to enter a filename to upload.

Line 7 performs one more check on the validity of the file selected by the user for uploading. It checks to ensure that the user has not typed in a filename that does not correspond to a file on his hard drive (by checking that the `ContentLength` property of the `PostedFile` property is greater than zero). Furthermore, line 7 checks to ensure that the user did not enter a blank filename in the file upload text box. Assuming that these conditions are met, the code from lines 10 through 19 is executed. This is the code that performs the actual work of saving the uploaded file to the Web server's file system.

On line 10 the constant `strBaseDir` is assigned the directory to save the uploaded files in. Next, on lines 11 and 12, the filename portion of the uploaded file is snipped out from the `FileName` property of the `PostedFile` property and assigned to the variable `strFileName`. If the user selected to upload the file `C:\Resume\MyResume.doc`, the `FileName` property would be equal to `C:\Resume\MyResume.doc`. Because we already know what directory in which we want to save the file, we simply want just the name of the file (without the path). The static version of the `GetFileNameFromPath` method of the `File` class can be used to quickly snare this portion of the `FileName` property (line 12).

Next, on line 14, the uploaded file is saved to the Web server's file system. The `SaveAs` method of the `HttpPostedFile` class expects a single parameter: the directory to upload the file to. Finally, we want to display a message to the user letting him know that his file has been successfully uploaded. Lines 17 through 19 accomplish this task, providing a link to the uploaded file.

## NOTE

If you want the uploaded files to be Web accessible, you must upload them to a Web accessible directory. For this example, I created a virtual directory named `/Upload` that pointed to `C:\My Projects\Uploaded Files\`.

## Working Directly with an Uploaded File

Listing 2.23 illustrated how to upload a file from the client to the Web server and then save this file to the Web server's file system. This is useful for scenarios in which you want the user and, perhaps, other visitors to be able to access the uploaded content at a later date. But what if you only needed to perform some task with the uploaded data and then no longer needed it? Why bother saving it to the file system at all?

The `InputStream` property of the `HttpPostedFile` class provides direct access to the contents of the uploaded file in the form of a `Stream`. Therefore, a developer can programmatically access this information as opposed to saving the file first, if he so desires.

Listing 2.24 illustrates accessing an uploaded file via the `InputStream`. Listing 2.24 allows the user to upload an image file and then displays information about the image, such as its dimensions, resolution, and file type.

### LISTING 2.24 Uploaded Files Need not Be Saved to the Web Server's File System; Developers Can Programmatically Access the Uploaded File's Contents

---

```
1: <%@ Import Namespace="System.IO" %>
2: <%@ Import Namespace="System.Drawing" %>
3: <%@ Import Namespace="System.Drawing.Imaging" %>
4: <script language="VB" runat="server">
5: Sub btnSubmit_OnClick(source as Object, e as EventArgs)
6:   'Read in the uploaded file into a Stream
7:   Dim objStream as Stream = fupUpload.PostedFile.InputStream
8:   Dim objImage as System.Drawing.Image = _
9:       System.Drawing.Image.FromStream(objStream)
10:
11:   'List the properties of the Image
12:   Dim strFileName as String = fupUpload.PostedFile.FileName
13:   lblImageProps.Text = "<hr><p><b>Uploaded File:</b> " & strFileName & _
14:       "<br><b>Type:</b> "
15:
16:   'Determine the file type
17:   If objImage.RawFormat.Equals(ImageFormat.GIF) then lblImageProps.Text &=
18:       "GIF"
19:   If objImage.RawFormat.Equals(ImageFormat.BMP) then lblImageProps.Text &=
20:       "BMP"
21:   If objImage.RawFormat.Equals(ImageFormat.JPEG) then lblImageProps.Text &=
22:       "JPEG"
23:   If objImage.RawFormat.Equals(ImageFormat.Icon) then lblImageProps.Text &=
24:       "Icon"
25:   If objImage.RawFormat.Equals(ImageFormat.TIFF) then lblImageProps.Text &=
26:       "TIFF"
27: End Sub
```

**LISTING 2.24** Continued

```
23: lblImageProps.Text &= "<br><b>Dimensions:</b> " & objImage.Width & _
24:     " x " & objImage.Height & " pixels<br><b>Size:</b> " & _
25:     objStream.Length & " bytes<br>" & "<b>Horizontal Resolution:</b> " &
_
26:     objImage.HorizontalResolution & _
27:     " pixels per inch<br><b>Vertical Resolution:</b> " & _
28:     objImage.VerticalResolution & " pixels per inch" & _
29:     "<p><img src="" & strFileName & "">"
30:
31:     objImage.Dispose()
32: End Sub
33: </script>
34:
35: <html>
36: <body>
37:     <form runat="server" EncType="multipart/form-data">
38:         <h1>Image Converter</h1>
39:         <b>Select an image file:</b><br>
40:         <input runat="server" id="fupUpload" type="file">
41:         <p>
42:         <asp:button id="btnSubmit" runat="server" Text="Get Image Info"
43:             OnClick="btnSubmit_OnClick" />
44:
45:         <p><asp:label id="lblImageProps" runat="server" />
46:     </form>
47: </body>
48: </html>
```

Listing 2.24 begins, as many of our examples have thus far, with a number of `Import` directives. Because we will be dealing with the `Stream` class, the `System.IO` namespace is imported (line 1). Likewise, because Listing 2.24 will need to be able to determine an image's various properties, we will need to work with the `Image` and `ImageFormat` classes, which require the `System.Drawing` and `System.Drawing.Imaging` namespaces, respectively.

Because Listing 2.24 does not contain the `Page_Load` event handler, let's begin our examination of the code starting with the HTML content (lines 35 through 48). (This HTML content is very similar to the HTML content presented in Listing 2.23.) We start by creating a post-back form that contains the always important `enctype="multipart/form-data"` property (line 37). Next, a file upload HTML control is created on line 40. A button control, `btnSubmit`, is created on lines 42 and 43; note that the button's `OnClick` event is wired up to the `btnSubmit_OnClick` event handler, which we'll examine shortly. Finally, on line 45, there is a label control that will be used to display the uploaded image file's properties.

The HTML content defined from line 35 through line 48 will present the user with an upload form field (a text box with the accompanying Browse button). After the user selects a file and clicks the Get Image Info button, the form will be submitted, the selected image file will be uploaded to the server, and the `btnSubmit_OnClick` event handler will fire, displaying the properties of the uploaded image file.

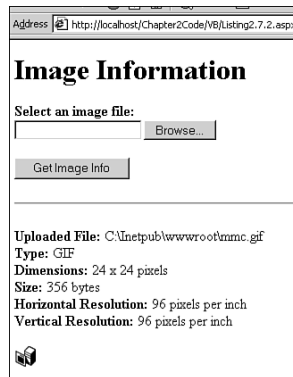
The `btnSubmit_OnClick` event handler begins on line 5 and starts by assigning a `Stream` variable, `objStream`, to the `Stream` instance returned by the `InputStream` property (line 7). At this point, the contents of the uploaded file are accessible via the `objStream` variable. Next, on line 8, an `Image` class variable is created and assigned the value returned from the `FromStream` method of the `Image` class. (Recall that the `Image` class is abstract, and therefore cannot be instantiated with the `new` keyword.)

Now that we have an `Image` class instance representing the uploaded image file, we can go ahead and list its properties. On line 12 we assign the `FileName` property of the `PostedFile` property to the variable `strFileName`. Next, this variable is displayed in the `lblImageProps` label control (lines 13 and 14).

Next, on lines 16 through 21, we check to determine the file type of the uploaded image. The `RawFormat` property of the `Image` class returns an instance of the `ImageFormat` class that represents the file format of the image. The `Equals` method is a method of the `Image` class (inherited from the `Object` class) that checks to determine if two `Objects` are equivalent. We check the return value of the `RawFormat` property against the various `ImageFormat` types to determine the file type of the upload image.

In lines 23 through 29, we output the various properties of the `Image` class: the `Height`, `Width`, file size (`Length` property of the `Stream` class), `HorizontalResolution`, and `VerticalResolution`. We also output an `IMG` tag with the `SRC` property specifying the path to the image on the client's computer. Finally, on line 31, we clean up the resources claimed by the `objImage` class.

Figure 2.25 shows the output of Listing 2.24 after a user has selected an image file from his hard drive. Keep in mind that the uploaded image file is never saved to the Web server's file system.

**FIGURE 2.25**

*This ASP.NET page provides information about an image on the user's hard drive.*

## Using ProcessInfo: Retrieving Information About a Process

One of the biggest benefits of ASP.NET over classic ASP, in my opinion, is ASP.NET's capability to automatically restart itself based on programmatically defined conditions. With classic ASP, developers often felt the need to reboot the Web server once a day (or week, or whatever) to ensure that things chunked along smoothly, free of any dreaded Server Too Busy errors.

Not only does ASP.NET allow the developer to set conditions on when to have the Web service restarted, but it also provides access to information on both the state of the currently running `xspwp.exe` process and previously running instances! (The `xspwp.exe` process is the process that is responsible for handling calls to ASP.NET pages, similar to `asp.dll` with classic ASP.) This means that, through an ASP.NET page, you can determine your Web site's uptime, find out why the `xspwp.exe` process might have restarted itself, or see how much memory the current `xspwp.exe` process has consumed.

All these tasks are handled through a pair of classes: `ProcessInfo` and `ProcessModelInfo`. The `ProcessInfo` class handles information about a specific instance of the `xspwp.exe` process, whereas the `ProcessModelInfo` class provides access to information on both current and past instances of the process.

## Displaying Information for the Currently Executing `xspwp.exe` Process

Listing 2.25 shows how to use these two classes to retrieve information about the currently executing `xspwp.exe` process. Because this ASP.NET page returns information such as

memory consumed by the `xspwp.exe` process and the current uptime, it could be as a monitoring page for the site's Web master.

### LISTING 2.25 Gather Information on the Web Site's Uptime Through an ASP.NET Page

```
1: <script language="VB" runat="server">
2: Sub Page_Load(source as Object, e as EventArgs)
3: 'Display information about the xspwp.exe process
4: Dim objProcInfo as ProcessInfo = ProcessModelInfo.GetCurrentProcessInfo()
5:
6: lblStartTime.Text = objProcInfo.StartTime
7: lblAge.Text = Single.Format(objProcInfo.Age.TotalHours, "#.##") & "
hours"
8: Select Case objProcInfo.Status
9: Case ProcessStatus.Alive:
10: lblStatus.Text = "Alive"
11: Case ProcessStatus.ShutDown:
12: lblStatus.Text = "Shut down"
13: Case ProcessStatus.ShuttingDown:
14: lblStatus.Text = "Currently shutting down"
15: Case ProcessStatus.Terminated:
16: lblStatus.Text = "Terminated"
17: End Select
18:
19: lblProcessID.Text = objProcInfo.ProcessID
20: lblPeakMemoryUsed.Text = Int32.Format(objProcInfo.PeakMemoryUsed,
"#,###") & " K"
21: End Sub
22: </script>
23:
24: <html>
25: <body>
26: <table align="center" border="1" cellspacing="0" cellpadding="5">
27: <tr>
28: <th colspan=2>
29: <code>xspwp.exe</code> Process Information
30: </th>
31: </tr>
32: <tr>
33: <td>Start Time</td>
34: <td><asp:label runat="server" id="lblStartTime" /></td>
35: </tr>
36: <tr>
37: <td>Age</td>
38: <td><asp:label runat="server" id="lblAge" /></td>
39: </tr>
```

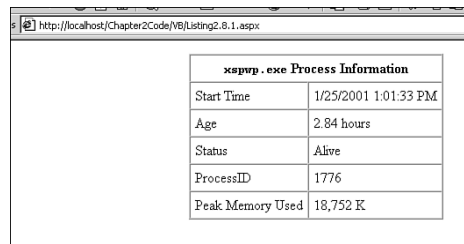


**LISTING 2.25** Continued

```

40:     <tr>
41:         <td>Status</td>
42:         <td><asp:label runat="server" id="lblStatus" /></td>
43:     </tr>
44:     <tr>
45:         <td>ProcessID</td>
46:         <td><asp:label runat="server" id="lblProcessID" /></td>
47:     </tr>
48:     <tr>
49:         <td>Peak Memory Used</td>
50:         <td><asp:label runat="server" id="lblPeakMemoryUsed" /></td>
51:     </tr>
52: </table>
53: </body>
54: </html>

```



xspwp.exe Process Information	
Start Time	1/25/2001 1:01:33 PM
Age	2.84 hours
Status	Alive
ProcessID	1776
Peak Memory Used	18,752 K

**FIGURE 2.26**

Information about the currently executing xspwp.exe process is displayed.

The code in Listing 2.25 is fairly straightforward. It simply grabs the information about the currently executing xspwp.exe process and displays its information in an HTML table. To accomplish this, we must first create an instance of the `ProcessInfo` class (line 4). To get information about the currently executing xspwp.exe process, though, we must make a call to the `GetCurrentProcessInfo` method of the `ProcessModelInfo` class (line 4). After line 4 has executed, the `objProcInfo` variable has been assigned to a `ProcessInfo` instance populated with the information on the currently executing xspwp.exe process.

Lines 7 through 20 simply display the various properties of the `ProcessInfo` class. On line 6, the start time of the currently executing xspwp.exe process is displayed; next, on line 7, the number of hours the process has been running is displayed. The `Status` property of the `ProcessInfo` class returns a `ProcessStatus` enumeration representing the status of the currently executing process. On lines 8 through 17, a `Select Case` is used to display an English representation of this numeric property. Finally, on lines 19 and 20, the `ProcessID` and `PeakMemoryUsed` properties are displayed. The HTML from lines 24 through 54 simply display

an HTML table with a number of label controls to represent the various properties of the `ProcessInfo` class.

## Displaying Information for Past Instances of the `xspwp.exe` Process

The `ProcessModelInfo` has another handy method, `GetHistory`, which returns information on a specified number of past instances of the `xspwp.exe` process. This means that you can, after a Web service restart, view the cause of the restart from an ASP.NET page! The `GetHistory` method expects one parameter, an integer representing the number of past instances to return; the method returns an array of `ProcessInfo` instances.

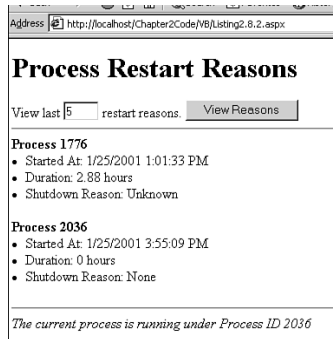
Listing 2.26 provides an ASP.NET page in which the user can enter how many Web service restarts back he wants to view. The past (and current) processes are listed displaying when they started, how long they ran for, and why they were shut down. As with Listing 2.26, this ASP.NET page would make a very handy tool for a Web site's administrator. The output is shown in Figure 2.27.

### LISTING 2.26 The `ProcessModelInfo` Class Can Provide Information on Past Instances of the `xspwp.exe` Process

```
1: <script language="VB" runat="server">
2:   Sub Page_Load(source as Object, e as EventArgs)
3:     Dim iCount as Integer = 10
4:
5:     If Not Page.IsPostBack Then
6:       'View the last 10 restart reasons
7:       txtCount.Text = iCount
8:     Else
9:       'View the recounts defined by txtCount
10:      iCount = txtCount.Text.ToInt32()
11:    End If
12:
13:    'Display the reasons for the last iCount restarts
14:    Dim aProcInfos() as ProcessInfo = ProcessModelInfo.GetHistory(iCount)
15:    Dim objProcInfo as ProcessInfo
16:
17:    lblProcHistory.Text = ""
18:    For Each objProcInfo in aProcInfos
19:      lblProcHistory.Text &= "<b>Process " & objProcInfo.ProcessID & _
20:        "</b><br><li>Started At: " & objProcInfo.StartTime & "<br>" & _
21:        "<li>Duration: " & _
22:        Single.Format(objProcInfo.Age.TotalHours, "0.##") & " hours<br>" &
```

**LISTING 2.26** Continued

```
23:         "<li>Shutdown Reason: " &
DisplayReason(objProcInfo.ShutdownReason) & _
24:         "<p>"
25:     Next
26:
27:     Dim objCurrentProc as ProcessInfo =
ProcessModelInfo.GetCurrentProcessInfo()
28:     lblCurrentProc.Text = "<i>The current process is running under Process
ID " & _
29:         objCurrentProc.ProcessID & "</i>"
30: End Sub
31:
32: Function DisplayReason(enumReason as ProcessShutdownReason) as String
33:     Select Case enumReason
34:         Case ProcessShutdownReason.IdleTimeout:
35:             DisplayReason = "Idle Timeout"
36:         Case ProcessShutdownReason.MemoryLimitExceeded:
37:             DisplayReason = "Memory Limit Exceeded"
38:         Case ProcessShutdownReason.None:
39:             DisplayReason = "None"
40:         Case ProcessShutdownReason.RequestQueueLimit:
41:             DisplayReason = "Request Queue Limit"
42:         Case ProcessShutdownReason.RequestsLimit:
43:             DisplayReason = "Requests Limit"
44:         Case ProcessShutdownReason.Timeout:
45:             DisplayReason = "Timeout"
46:         Case ProcessShutdownReason.Unexpected:
47:             DisplayReason = "Unknown"
48:     End Select
49: End Function
50: </script>
51:
52: <html>
53: <body>
54:     <form runat="server">
55:         <h1>Process Restart Reasons</h1>
56:         View last
57:         <asp:textbox id="txtCount" runat="server" Size="3" />
58:         restart reasons.
59:         <asp:button id="btnSubmit" runat="server" Text="View Reasons" />
60:         <hr>
61:         <asp:label runat="server" id="lblProcHistory" />
62:         <hr><asp:label id="lblCurrentProc" runat="server" />
63:     </form>
64: </body>
65: </html>
```

**FIGURE 2.27**

*View information on the last N process invocations of xspwp.exe.*

Listing 2.26 begins by determining how many Web service restarts back to view information on. The variable `iCount` represents how many Web service restarts back to display. If the page is loaded for the first time, `iCount` will have a value of `10`. Otherwise, `iCount` will be set equal to the value specified by the user in the `txtCount` text box.

Next, on line 14, an array of `ProcessInfo` instances are retrieved by the `GetHistory` method of the `ProcessModelInfo` class. This array of `ProcessInfo` instances is then assigned to the variable `aProcInfos`. On lines 18 through 25, a `For Each ... Next` loop is used to iterate through each element of the array `aProcInfos`. Similar to the `Status` property we examined in Listing 2.25, the `ShutdownReason` property of the `ProcessInfo` class is an enumeration. Therefore, rather than displaying the numerical representation of the enumeration, we use a helper function, `DisplayReason`, that will provide an English-readable explanation of why the previous Web service restart occurred.

The HTML content, spanning lines 52 through 65, creates a post-back form with a text box (line 57), a submit button (line 59), and a label control to display the process information (line 62). The text box, `txtCount`, allows the user to enter an exact number of past `xspwp.exe` processes to review. When the button `btnSubmit` is clicked, the form is posted back; the `Page_Load` event handler will fire and set `iCount` to the value of `txtCount` (see line 10).

## Accessing the Windows Event Log

Hopefully, by now you realize that there are a plethora of tasks you can do with ASP.NET which, with classic ASP, were impossible or extremely difficult without the use of a COM component. One such task that was not possible with just classic ASP was the ability to access the Windows event log. The .NET Framework, however, contains a number of classes that allow developers to both read and write to the event log!

This means that you can create an ASP.NET page through which a user could view event log history. Such functionality, coupled with the `ProcessInfo` ASP.NET page discussed in “Using `ProcessInfo`: Retrieving Information about a Process,” could be used to create a very powerful remote administration tool for site information and maintenance purposes. Additionally, because an ASP.NET page can write entries to the event log, you could have various ASP.NET errors recorded to the Windows event log.

This section is broken down into two parts. In the first part, “Reading from the Event Log,” we’ll look at an ASP.NET page that displays the current entries from a selected log. In the second part, “Writing to the Event Log,” we’ll examine how to record events to the event log.

## Reading from the Event Log

The .NET Framework contains a number of classes to read and write to the event log, all of which reside in the `System.Diagnostics` namespace. The main class that provides access to the event log is aptly named `EventLog`. Keep in mind that the Windows event log can contain many logs itself. Windows 2000 comes with three default logs: the Application log, the Security log, and the System log. The `EventLog` class can read and write to any of these various logs. In fact, the `EventLog` class can even be used to create new logs in the Windows event log.

### NOTE

The `EventLog` class can be used to read and write to remote computers’ Windows event logs as well as for the Web server’s Windows event log. For more information on accessing remote event logs, refer to the documentation for the `EventLog` class.

Listing 2.27 shows a very simple ASP.NET page that displays the event log entries in the System event log that have been registered as errors. The output of Listing 2.27, when viewed through a browser, can be seen in Figure 2.27.

### LISTING 2.27 The `EventLog` Class Provides Access to the Windows Event Log

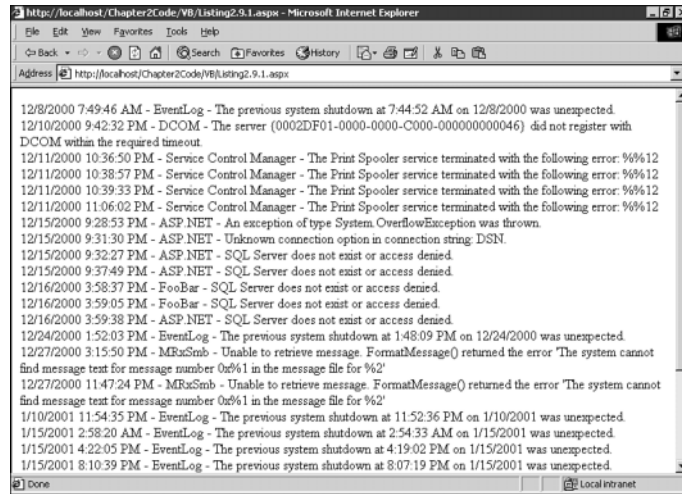
```
1: <%@ Import Namespace="System.Diagnostics" %>
2: <script language="VB" runat="server">
3:     Sub Page_Load(source as Object, e as EventArgs)
4:         Dim objEventLog as EventLog = New EventLog("System")
5:
6:         Dim objEntry as EventLogEntry
7:         For Each objEntry in objEventLog.Entries
8:             If objEntry.EntryType = EventLogEntryType.Error then
9:                 Response.Write(objEntry.TimeGenerated & " - " & _
10:                    objEntry.Source & " - " & _
```

**LISTING 2.27** Continued

```

11:         objEntry.Message & "<br>"
12:     End If
13:     Next
14: End Sub
15: </script>

```

**FIGURE 2.28**

*Displays the error entries in the System log.*

Listing 2.27 begins with an `Import` directive to include the namespace of the `EventLog` class, `System.Diagnostics`. In the `Page_Load` event handler, an instance of the `EventLog` class, `objEventLog`, is created (line 4). There are many forms of the `EventLog` constructor; on line 4, we used the one that expects a single `String` parameter specifying the log filename to open.

Each log in the Windows event log is composed of a number of entries. The .NET Framework provides an abstraction of each event log entry as a class, `EventLogEntry`. The `EventLog` class contains an `Entries` property that exposes a collection of `EventLogEntry` instances representing all the entries for a specific event log. On lines 7 through 13, we iterate through this collection using a `For Each ... Next` loop (line 7).

The `EventLogEntry` class contains a number of properties that represent an entry in the event log. One of these properties is `EntryType`, which indicates the type of event log entry. The possible values for this property are defined by the `EventLogEntryType` enumeration, and include values such as `Error`, `Information`, `Warning`, and others. One line 8, we check to determine if the current `EventLogEntry` instance in our `For Each ... Next` loop is an error entry. If it is,

we display additional information about the entry (lines 9 through 11); otherwise, we skip on to the next entry.

Although viewing all the error entries for a particular log file might have plausible uses, we could easily create a much more graphically pleasing and easier to use interface without much effort. Listing 2.28 contains the code for an ASP.NET page that lists all the entries for a user-chosen log file. Furthermore, these event log entries are displayed through the creation of a dynamic HTML table using the various ASP.NET table controls. Figure 2.29 contains a screenshot of Listing 2.28 when viewed through a browser.

**LISTING 2.28** The Complete Contents of a User-Chosen Event Log Are Displayed with Pretty Formatting

```
1: <%@ Import Namespace="System.Diagnostics" %>
2: <%@ Import Namespace="System.Drawing" %>
3: <script language="VB" runat="server">
4:   Sub Page_Load(source as Object, e as EventArgs)
5:     If Not Page.IsPostBack Then
6:       DisplayEventLog("System")
7:     End If
8:   End Sub
9:
10:  Sub btnSubmit_OnClick(source as Object, e as EventArgs)
11:    DisplayEventLog(1stLog.SelectedItem.Value)
12:  End Sub
13:
14:  Sub btnClear_OnClick(source as Object, e as EventArgs)
15:    'Clear all of the event log entries
16:    Dim objEventLog as New EventLog(1stLog.SelectedItem.Value)
17:    objEventLog.Clear()
18:  End Sub
19:
20:  Sub DisplayEventLog(strLogName as String)
21:    Dim objRow as New TableRow
22:    Dim objCell as New TableCell
23:
24:    objCell.BackColor = Color.Bisque
25:    objCell.HorizontalAlign = HorizontalAlign.Center
26:    objCell.Text = "Type"
27:    objRow.Cells.Add(objCell)
28:
29:    objCell = New TableCell
30:    objCell.BackColor = Color.Bisque
31:    objCell.HorizontalAlign = HorizontalAlign.Center
32:    objCell.Text = "Date"
```

**LISTING 2.28** Continued

```
34:
35:     objCell = New TableCell
36:     objCell.BackColor = Color.Bisque
37:     objCell.HorizontalAlign = HorizontalAlign.Center
38:     objCell.Text = "Time"
39:     objRow.Cells.Add(objCell)
40:
41:     objCell = New TableCell
42:     objCell.BackColor = Color.Bisque
43:     objCell.HorizontalAlign = HorizontalAlign.Center
44:     objCell.Text = "Source"
45:     objRow.Cells.Add(objCell)
46:
47:     objCell = New TableCell
48:     objCell.BackColor = Color.Bisque
49:     objCell.HorizontalAlign = HorizontalAlign.Center
50:     objCell.Text = "User"
51:     objRow.Cells.Add(objCell)
52:
53:     objCell = New TableCell
54:     objCell.BackColor = Color.Bisque
55:     objCell.HorizontalAlign = HorizontalAlign.Center
56:     objCell.Text = "Computer"
57:     objRow.Cells.Add(objCell)
58:
59:     tblLog.Rows.Add(objRow)
60:
61:
62:     Dim objEventLog as EventLog = New EventLog(strLogName)
63:     Dim objEntry as EventLogEntry
64:
65:     For Each objEntry in objEventLog.Entries
66:         objRow = New TableRow
67:         objCell = New TableCell
68:
69:         'Determine the type of error
70:         If objEntry.EntryType = EventLogEntryType.Error Then
71:             objCell.BackColor = Color.Red
72:             objCell.ForeColor = Color.White
73:             objCell.Text = "Error"
74:         ElseIf objEntry.EntryType = EventLogEntryType.Information Then
75:             objCell.Text = "Information"
76:         ElseIf objEntry.EntryType = EventLogEntryType.Warning Then
77:             objCell.BackColor = Color.Yellow
78:             objCell.Text = "Warning"
```



**LISTING 2.28** Continued

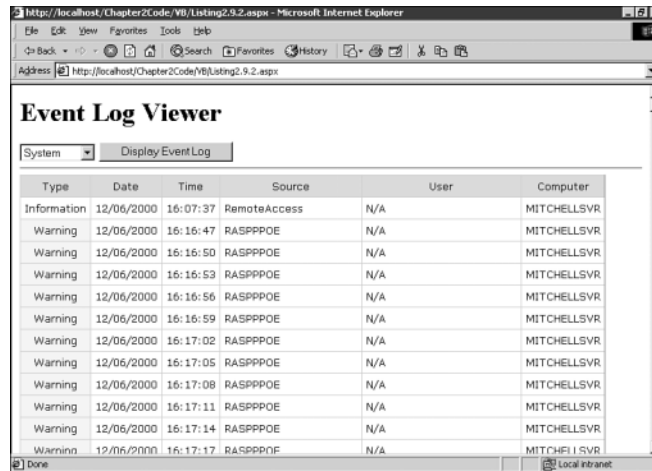
```
79:         ElseIf objEntry.EntryType = EventLogEntryType.SuccessAudit Then
80:             objCell.Text = "Success Audit"
81:         ElseIf objEntry.EntryType = EventLogEntryType.FailureAudit Then
82:             objCell.ForeColor = Color.Red
83:             objCell.Text = "Failure Audit"
84:         End If
85:         objCell.HorizontalAlign = HorizontalAlign.Center
86:         objRow.Cells.Add(objCell)
87:
88:         objCell = New TableCell
89:         objCell.Text = objEntry.TimeGenerated.ToShortDateString()
90:         objRow.Cells.Add(objCell)
91:
92:         objCell = New TableCell
93:         objCell.Text = objEntry.TimeGenerated.ToLongTimeString()
94:         objRow.Cells.Add(objCell)
95:
96:         objCell = New TableCell
97:         objCell.Text = objEntry.Source
98:         objRow.Cells.Add(objCell)
99:
100:        objCell = New TableCell
101:        If objEntry.UserName <> Nothing then
102:            objCell.Text = objEntry.UserName
103:        Else
104:            objCell.Text = "N/A"
105:        End If
106:        objRow.Cells.Add(objCell)
107:
108:        objCell = New TableCell
109:        objCell.Text = objEntry.MachineName
110:        objRow.Cells.Add(objCell)
111:
112:
113:        tblLog.Rows.Add(objRow)
114:    Next
115: End Sub
116: </script>
117:
118: <html>
119: <body>
120:     <form runat="server">
121:         <h1>Event Log Viewer</h1>
122:         <asp:listbox runat="server" id="lstLog" Rows="1">
123:             <asp:listitem>Application</asp:listitem>
124:             <asp:listitem>Security</asp:listitem>
```

**LISTING 2.28** Continued

```

125:     <asp:listitem Selected="True">System</asp:listitem>
126: </asp:listbox>
127: <asp:button runat="server" id="btnSubmit" Text="Display Event Log"
128:         OnClick="btnSubmit_OnClick" />
129: <hr>
130: <asp:table runat="server" id="tblLog" CellPadding="5"
131:         CellSpacing="0" GridLines="Both" Font-Size="10pt"
132:         Font-Name="Verdana" />
133: <hr>
134: <asp:button runat="server" id="btnClear" Text="Clear Event Log"
135:         OnClick="btnClear_OnClick" />
136: </form>
137: </body>
138: </html>

```

**FIGURE 2.29**

Listing 2.28 displays an event log's entries through an ASP.NET page.

Listing 2.28 begins by importing two namespaces: `System.Diagnostics`, for the `EventLog` and `EventLogEntry` classes (line 1); and `System.Drawing`, for the `Color` structure that is used to set the foreground and background colors of the table cells (line 2). When the page is loaded for the first time, the `Page_Load` event handler will fire and line 6 will be reached, in which case the `DisplayEventLog` subroutine will be called to display the entries in the System log.

The `DisplayEventLog` subroutine, spanning from line 20 through line 115, displays all the entries for the event log specified by `strLogName` in a nice format. This format is in the form of an HTML table and created via the ASP.NET table controls (the `Table`, `TableRow`, and

TableCell Web controls). (For more information on these controls, refer to the .NET Framework Documentation.)

The HTML section for the page, found in lines 118 through 138, start by defining a post-back form (line 120). Next, a list box control is created on line 122 and hard-coded with the three default Windows 2000 event logs: Application, Security, and System (lines 123 through 125). Next, a button, `btnSubmit`, is created on lines 127 and 128. This button, when clicked, will post-back the form and cause the `btnSubmit_OnClick` event handler to fire. Next, on lines 130, 131, and 132, a table control is created. This is the table control that is dynamically built-up in the `DisplayEventLog` subroutine. Finally, on lines 134 and 135, another button control is created. When clicked, this button, `btnClear`, will cause all the event log entries for the selected event log to be cleared.

The `btnSubmit_OnClick` event handler can be found from lines 10 through 12. It is very simple, containing only one line of code. All it needs to do is display the event log selected by the user. This is accomplished by calling the `DisplayEventLog` subroutine and passing the `Value` of the selected list box item (line 11).

The `btnClear_OnClick` event handler is called when the `btnClear` button is clicked (see lines 134 and 135). This event handler is responsible for clearing all the event log entries, which is accomplished with the `Clear` method of the `EventLog` class (line 17).

## Writing to the Event Log

The `EventLog` class also provides the ability for developers to have information written to the event log. There are a number of real-world situations in which it might be advantageous for an ASP.NET page to be able to make an entry to the Windows event log. In an article on ASPFree.com (<http://aspfree.com/asp+/eventlog2.aspx>), Steve Schofield demonstrates how to create an error-handling page so that no matter when an ASP.NET error occurs, it is logged to a custom created error log.

Listing 2.29 provides similar functionality to Steve Schofield's event log article. Rather than recording any error in any ASP.NET Web page, however, Listing 2.29 provides a simple function that can be included in certain ASP.NET pages and called when an error occurs. Furthermore, rather than writing errors to a custom event log, Listing 2.29 records errors to the System event log.

**LISTING 2.29** With the .NET Framework, a Developer Can Add Entries to an Event Log via an ASP.NET Page

```
1: <%@ Import Namespace="System.Data" %>
2: <%@ Import Namespace="System.Data.SQL" %>
3: <%@ Import Namespace="System.Diagnostics" %>
```

**LISTING 2.29** Continued

---

```
4: <script language="c#" runat="server">
5:   void Page_Load(Object source, EventArgs e)
6:   {
7:     // Perform some type of illegal operation
8:     try {
9:       SqlConnection objConn;
10:      objConn = new
11:      SqlConnection("server=localhost;uid=foo;pwd=bar;database=pubs");
12:      objConn.Open();
13:      // ...
14:    }
15:    catch (Exception eError)
16:    {
17:      RecordError(eError, EventLogEntryType.Error);
18:    }
19:  }
20:
21: void RecordError(Exception eError, EventLogEntryType enumType)
22: {
23:   const String strSource = "ASP.NET",
24:   strLogName = "System";
25:
26:   // Add the strMessage entry to the ASPX event log
27:   EventLog objLog = new EventLog(strLogName);
28:   objLog.Source = strSource;
29:   objLog.WriteEntry(eError.Message, enumType);
30: }
31: </script>
```

---

Listing 2.29 demonstrates how an entry can be written to the event log when an error occurs. Although the code in the try block from lines 9 through 11 might seem valid, the connection string specified on line 10 is not. This will cause an error to be thrown on line 11, when attempting to open the connection. At this point, the catch block will catch the exception and make a call to the RecordError function, which will add an entry to the System event log indicating the error (line 16).

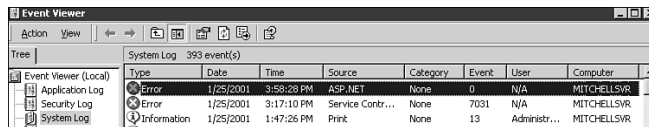
The RecordError function, stretching from line 21 through line 30, adds an entry to the System event log. First, on lines 23 and 24, two constants are defined: strSource, specifying the source of the error, and strLogName, specifying the event log that is to record the error. Next, on line 27, an EventLog instance representing the System event log is created. On line 28, the Source property is set to the constant strSource. Finally, on line 29, an entry is added to the event log. This entry's Message is identical to the Message property of the thrown

Exception. The entry type is specified by the developer, passed in as the second parameter to the `RecordError` function. In Listing 2.29, an `Error` entry type is specified (see line 16).

### NOTE

The Source of the entry added to the event log is used to specify the application or service that caused the error.

Figure 2.30 displays the detailed information for the entry added to the System event log for the error caused by Listing 2.28. Note that the event log entry has its source set to "ASP.NET" and indicates that there was an error in accessing the specified database. The `RecordError` function could be improved to add more detailed error messages, such as the ASP.NET page that threw the error and other relevant information.



**FIGURE 2.30**

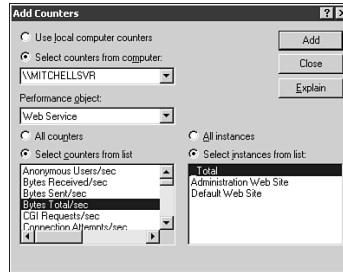
*A new error entry has been added to the System event log.*

## Working with Server Performance Counters

As we examined in “Accessing the Windows Event Log,” the .NET Framework contains a `System.Diagnostics` namespace that houses classes which can be used for diagnostic information. One such class is the `EventLog` class, which we just looked at in the previous section; another useful diagnostics class is the `PerformanceCounter` class. This class gives the developer direct read access to the existing Windows Performance counters and provides the capability to create new performance counters!

Because this functionality is encapsulated in the .NET Framework, these techniques can be used through an ASP.NET page. That means you can create an ASP.NET page that reports information on the Web server’s various resources! Talk about a remote administrator’s dream!

Performance counters, which are represented by the `PerformanceCounter` class, are defined by at least two required properties. The first two required properties are `CategoryName` and `CounterName`; these represent the category and counter one sees when selecting a performance monitor through the Administrative Performance tool (see Figure 2.31). For example, to view the percentage of the processor’s power being used, you’d be interested in the "% Processor Time" counter of the "Processor" category.

**FIGURE 2.31**

When choosing a performance counter through the administrative interface, a category and counter are selected.

For certain performance counters, a third property, `InstanceName`, is required. Specifically, the `InstanceName` is needed for counters that monitor a metric with multiple instances, such as Web server information. Because a single Web server can run many different Web sites, we must specify the Web site if we want to view specific Web server–related information (such as total number of requests).

When creating an instance of the `PerformanceCounter`, you can specify these two (or three) properties in the constructor. For example, to create an instance of the `PerformanceCounter` class that could be used to report the total number of threads, the following code could be used:

```
Dim objPerf as New PerformanceCounter("System", "Threads")
```

The previous snippet creates an instance of the `PerformanceCounter` class named `objPerf`. This instance represents the performance counter "Threads" found in the "System" category.

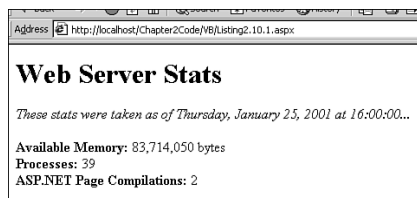
To read the performance counter's value, simply use the `NextValue()` method. `NextValue()` returns a single representing the value of the next performance counter's sample. Therefore, we could amend our previous code to display the total number of threads with a simple call to `NextValue()`:

```
Dim objPerf as New PerformanceCounter("System", "Threads")
Response.Write("There are " & objPerf.NextValue() & " threads.")
```

Listing 2.30 illustrates a very simple ASP.NET page that could possibly serve as a handy tool for an off-site Web site administrator. The code for Listing 2.30, when viewed through a browser, will display the total available memory for the Web server, the number of processes currently running on the Web server, and the total number of ASP.NET page compilations for the particular Web application in which the ASP.NET page resides. The output is shown in Figure 2.32.

**LISTING 2.30** Gather System Information about the Web Server via an ASP.NET Page

```
1: <%@ Import Namespace="System.Diagnostics" %>
2: <script language="VB" runat="server">
3: Sub Page_Load(source as Object, e as EventArgs)
4:   Dim objMemPerf as New PerformanceCounter("Memory", "Available Bytes")
5:   lblFreeMemory.Text = Single.Format(objMemPerf.NextValue(), "#,###") & "
bytes"
6:
7:   Dim objProcPerf as New PerformanceCounter("System", "Processes")
8:   lblProcesses.Text = objProcPerf.NextValue().ToString()
9:
10:  Dim strAppName as String = Request.ServerVariables("APPL_MD_PATH")
11:  strAppName = strAppName.Replace("/", "_")
12:  Dim objCompPerf as New PerformanceCounter("ASP Plus Applications", _
13:                                           "Page Compilations", strAppName)
14:
15:  lblPageComps.Text = objCompPerf.NextValue().ToString()
16: End Sub
17: </script>
18:
19: <html>
20: <body>
21:   <h1>Web Server Stats </h1>
22:   <i>These stats were taken as of <%=Now().ToLongDateString()%> at
23:   <%=Now().ToLongTimeString()%>...</i><p>
24:
25:   <b>Available Memory:</b> <asp:label id="lblFreeMemory" runat="server"
/><br>
26:   <b>Processes:</b> <asp:label id="lblProcesses" runat="server" /><br>
27:   <b>ASP.NET Page Compilations:</b>
28:     <asp:label id="lblPageComps" runat="server" /><br>
29: </body>
30: </html>
```

**FIGURE 2.32**

Performance counter values can be displayed through an ASP.NET page.

In Listing 2.30 we begin by importing the `System.Diagnostics` namespace (line 1). Next, in the `Page_Load` event handler, three `PerformanceCounter` class instances are created. The first one, `objMemPerf`, determines the amount of free memory in bytes available (line 4). On line 5, this value is outputted into the `lb1FreeMemory` label control. The `objProcPerf` performance counter on line 7 determines the total number of currently running processes and displays this value in the `lb1Processes` label control (line 8).

The third performance counter displays the total number of ASP.NET page compilations (lines 12 and 13). For this metric, we must specify what Web application to use. To accomplish this, we can either hard-code a value for the `InstanceName` property, or we can dynamically determine it through the `APPL_MD_PATH` server variable. This server variable has values such as `/LM/W3SVC/1/Root`, for example. The `PerformanceCounter` class, however, prefers the `InstanceName` in the following format: `_LM_W3SVC_1_Root`, so, on line 11, all the forward slashes (`/`) are replaced with underscores (`_`). Finally, on line 15, the value of the page compilations performance counter is outputted.

Lines 19 through 30 define the HTML content of the page. Lines 22 and 23 output the current system time that the stats were monitored, whereas lines 25, 27, and 28 create the three performance counter label controls. These three label controls are populated with the values from their respective `PerformanceCounter` instances in the `Page_Load` event handler.

Although being able to view a particular performance counter through an ASP.NET page is an amazing feat in itself, it's only the tip of the iceberg! The .NET Framework provides an additional class, `PerformanceCounterCategory`, that can be used to iterate through the Web server's available performance categories and counters. This means that you can provide a remote administrator with an ASP.NET page that lists the entire contents set of performance counters for the administrator to choose from. After a performance counter has been selected, its current value can be displayed.

Listing 2.31 illustrates how to use the `PerformanceCounterCategory` class—in conjunction with the `PerformanceCounter` class—to list all the available performance categories and counters, permitting the user to select a particular category and counter and view its current value. The output of Listing 2.31, when viewed through a browser, can be seen in Figure 2.33.

---

**LISTING 2.31** Iterate Through the Available Performance Categories and Counters with the `PerformanceCounterCategory` Class

---

```
1: <%@ Import Namespace="System.Diagnostics" %>
2: <script language="c#" runat="server">
3:     void Page_Load(Object source, EventArgs e)
4:     {
5:         if (!Page.IsPostBack)
6:         {
```



**LISTING 2.31** Continued

```
7:         foreach (PerformanceCounterCategory objPerfCounterCat in
8:             PerformanceCounterCategory.GetCategories())
9:             lstCategories.Items.Add(new
ListItem(objPerfCounterCat.CategoryName));
10:
11:         //Add the counters for the first category
12:         PopulateCounters(PerformanceCounterCategory.GetCategories()[0]);
13:     }
14: }
15:
16: void PopulateCounters(PerformanceCounterCategory objPerfCat)
17: {
18:     // Populates the lstCounters list box with the counters for the
selected
19:     // performance category (objPerfCat)...
20:     lstCounters.Items.Clear();
21:
22:     foreach (PerformanceCounter objCounter in objPerfCat.GetCounters())
23:         lstCounters.Items.Add(new ListItem(objCounter.CounterName));
24:
25:     DisplayCounter(objPerfCat.GetCounters()[0]);
26: }
27:
28: void lstCategories_OnChange(Object source, EventArgs e)
29: {
30:     PerformanceCounterCategory objPerfCat;
31:     objPerfCat = new
PerformanceCounterCategory(lstCategories.SelectedItem.Value);
32:
33:     lstInstances.DataSource = objPerfCat.GetInstanceNames();
34:     lstInstances.DataBind();
35:
36:     PopulateCounters(objPerfCat);
37: }
38:
39: void lstCounters_OnChange(Object source, EventArgs e)
40: {
41:     // Display counter information for the selected counter and category
42:     if (lstCounters.SelectedItem != null)
43:     {
44:         PerformanceCounter objPerf = new PerformanceCounter();
45:         objPerf.CategoryName = lstCategories.SelectedItem.Value;
46:         objPerf.CounterName = lstCounters.SelectedItem.Value;
47:         objPerf.InstanceName = lstInstances.SelectedItem.Value;
48:
49:         DisplayCounter(objPerf);
```

**LISTING 2.31** Continued

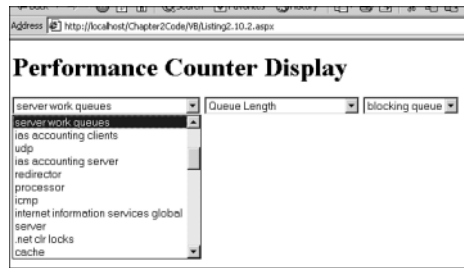
```
50:     }
51: }
52:
53: void lstInstances_OnChange(Object source, EventArgs e)
54: {
55:     // Display counter information for the selected counter and category
56:     if (lstCounters.SelectedItem != null)
57:     {
58:         PerformanceCounter objPerf = new PerformanceCounter();
59:         objPerf.CategoryName = lstCategories.SelectedItem.Value;
60:         objPerf.CounterName = lstCounters.SelectedItem.Value;
61:         objPerf.InstanceName = lstInstances.SelectedItem.Value;
62:
63:         DisplayCounter(objPerf);
64:     }
65: }
66:
67: void DisplayCounter(PerformanceCounter objPerf)
68: {
69:     try {
70:         objPerf.NextValue();
71:         lblLastValue.Text = objPerf.NextValue().ToString();
72:     }
73:     catch (Exception e)
74:     {
75:         try {
76:             PerformanceCounterCategory tmpCat = new
77:                 PerformanceCounterCategory(lstCategories.SelectedItem.Value);
78:             objPerf.InstanceName = tmpCat.GetInstanceNames()[0];
79:             objPerf.NextValue();
80:             lblLastValue.Text = objPerf.NextValue().ToString();
81:         }
82:         catch (Exception e2)
83:         {
84:             lblLastValue.Text = "<font color=red>ERROR: " + e.Message +
"</font>";
85:         }
86:     }
87: }
88: </script>
89:
90: <html>
91: <body>
92:     <form runat="server">
93:         <h1>Performance Counter Display</h1>
```

**LISTING 2.31** Continued

```

94:     <asp:listbox runat="server" id="lstCategories" Rows="1"
AutoPostBack="True"
95:         OnSelectedIndexChanged="lstCategories_OnChange" />
96:     <asp:listbox runat="server" id="lstCounters" Rows="1"
AutoPostBack="True"
97:         OnSelectedIndexChanged="lstCounters_OnChange" />
98:     <asp:listbox runat="server" id="lstInstances" Rows="1"
AutoPostBack="True"
99:         OnSelectedIndexChanged="lstInstances_OnChange" />
100:    <p>
101:        <b>Current Value:</b>
102:        <asp:label id="lblLastValue" runat="server" />
103:    </form>
104: </body>
105: </html>

```

**FIGURE 2.33**

A list of available performance categories and counters are presented to the user.

Before examining the script block in Listing 2.31, let's first turn our attention to the HTML section of the page (lines 90 through 105). Keep in mind the intended functionality for the code: to list the available performance categories and counters on the Web server. To accomplish this, we'll use three list boxes: the first list box will list the available performance categories; the second will list the available performance counters for the selected performance category; and the third will list the available instance values for the selected performance category.

These three list boxes are created on lines 94 through 99. The performance category list box, `lstCategories`, has the `OnSelectedIndexChanged` event wired up to the `lstCategories_OnChange` event handler (defined from lines 28 through 37). When this list box's currently selected item is changed, the form will be posted back and the `lstCategories_OnChange` event handler will fire. The list box for each counter, `lstCounters`, is defined on lines 96 and 97. This list box, too, has an event handler (`lstCounters_OnChange`) wired up to the `OnSelectedIndexChanged` event handler. Finally, `lstInstances`, the last list box, is defined on lines 98 and 99.

The final server control is a label control, defined on line 102. Its purpose is to display the current value of the selected performance counter. All these controls are encased by a post-back form control (defined on line 92).

The `Page_Load` event handler, found spanning lines 3 through 14, only executes code the first time the page is visited. This is because of the conditional on line 5 that checks to determine whether the form has been posted back. If it is the first visit to the page (that is, the form has not been posted back), all the available performance categories are iterated through. The `PerformanceCounterCategory` class contains a static method `GetCategories()`, which will return an array of `PerformanceCounterCategory` instances that represent the available performance categories on the machine. A `foreach` loop is used to iterate through all these categories, adding a new item to the `1stCategories` list box at each iteration (line 9). On line 12, the `PopulateCounters` function is called, which populates the `1stCounters` list box with the applicable counters for the selected category.

The `PopulateCounters` category begins by clearing out all the items in the `1stCounters` list box (line 20). Next, on line 22, a `foreach` loop is used to iterate through each of the counters in the performance category. The `GetCounters()` method returns all the available counters (see line 22). Each of these counter names are then added to the `1stCounters` list box (line 23). Before the `PopulateCounters` function terminates, it calls `DisplayCounter`, a function that displays the current value of the selected performance counter.

The `DisplayCounter` function (defined from lines 67 through 87) lists the current value of the passed-in `PerformanceCounter`, `objPerf`. A `try ... catch` block is used on line 69 to ensure that the performance counter can be successfully read. If an error occurs, we assume that it is because the instance name hasn't been specified. In such a case, the catch block starting on line 73 will begin executing, and another `try ... catch` block ensues (line 75). Here, the default instance name for the selected performance category is selected and used and the performance counter's value is outputted. The `GetInstanceNames()` method of the `PerformanceCounterCategory` class will return a `String` array of available instance names; on line 78 we simply choose the first available instance name. Again, if an error occurs, the catch block on line 82 will begin executing; at this point, we don't know how to handle the error, and the `Message` property of the exception is displayed (line 84). Because some performance counters require two consecutive reads to the `NextValue()` method to retrieve a meaningful value, two calls are made (lines 70 and 71 and lines 79 and 80).

Whenever the user changes the currently selected performance category in the `1stCategories` list box, the form is posted back and the `1stCategories_OnChange` event handler fires (see lines 28 through 37). This event handler creates a `PerformanceCounterCategory` instance representing the currently selected performance category (lines 30 and 31). Next, the `1stInstances` list box is populated with the available instance names for the categories using

data binding (lines 33 and 34). Finally, the `PopulateCounters` method is called (line 36), thereby populating the `1stCounters` list box with the appropriate counters for the selected performance category.

When the user selects a performance counter from the `1stCounters` list box, the form is posted back and the `1stCounters_OnChange` event handler is executed (see lines 39 through 51). This event handler begins by ensuring that a valid selection in the `1stCounters` list box has been made (line 42). Next, a `PerformanceCounter` instance is created (line 44), and its `CategoryName`, `CounterName`, and `InstanceName` properties are set based on the selected values in the corresponding list boxes. Finally, on line 49, the `DisplayCounter` method is called, displaying the current value for the selected performance counter. The `1stInstances_OnChange` event handler (lines 53 through 65), which fires when the user changes the current selection in the `1stInstances` list box, contains identical code to the `1stCounters_OnChange` list box.

To see Listing 2.31 in action, refer back to Figure 2.32. Using some enhancements we discussed earlier in this chapter (see the section “Generating Images Dynamically”), one could conceivably create an on-the-fly chart that would display the current (and past) status of one or more performance counters through an ASP.NET page, similar to the Performance tool in Windows NT/2000.

## Encrypting and Decrypting Information

With the .NET Framework, a vast number of cryptography protocols are presented for use in the form of classes. (All these classes can be found in the `System.Security.Cryptography` namespace.) A plethora of crypto protocols are embedded in the .NET Framework, and an in-depth discussion of cryptography or a comparison of the various protocols is far beyond the scope of this book. In this section, we will briefly discuss the difference between the two main types of cryptography (asymmetric and symmetric) and look at an example of encrypting and decrypting information using a symmetric crypto protocol.

### NOTE

With classic ASP, encrypting and decrypting information was possible through plain, vanilla script, but was best handled by a third-party component such as `ASPEncrypt` (see <http://www.ASPEncrypt.com/> for more information). To learn more about cryptography with classic ASP, check out <http://www.4guysfromrolla.com/Security/>.

In general terms, a cryptography protocol can be thought of as a black box that accepts two inputs—a textual input and a key—and returns textual output. To encrypt information, the plain-text message and a key are used as the input and the encrypted message is outputted.

To decrypt the information, the encrypted message and the key are entered as inputs and the decrypted (plain-text) message is returned.

Cryptography protocols come in two flavors, differing on how the key input is chosen. The first, and simplest, type of cryptography algorithm is referred to as symmetric cryptography. With symmetric cryptography, a single key is used for both encrypting and decrypting the information. Therefore, both the person who wants to encrypt information and the person who wants to decrypt the information must know this key. With symmetric cryptography, the single, universal key serves as a sort of password in which those who know the password can encrypt and decrypt information with ease; those who do not know the password are unable to either encrypt or (more importantly) decrypt information.

For example, with a symmetric cryptography algorithm, if Bob wanted to send Steve an encrypted message, Bob would take the plain-text message and encrypt it with a secret key that both he and Steve agreed upon. This encrypted information could then be shuttled to Steve who could decrypt it using the agreed upon key.

Symmetric cryptography has its disadvantages. Because both the receiver and sender of information must be privy to the key, how can the sender securely inform the receiver of the key being used? Asymmetric cryptography algorithms compensate for this weakness of symmetric cryptography algorithms. With asymmetric algorithms, each person has a set of personal keys: a private and a public key. An individual's private key is kept to himself, shared with no one, whereas everyone knows an individual's public key.

With asymmetric cryptography protocols, an individual's public and private keys are mathematically related in such a way that information encrypted with a public key can only be decrypted with the corresponding private key. This way, if Bob wants to send Steve an encrypted message, Bob can use Steve's public key to encrypt the information. After this is completed, the encrypted information can only be decrypted using Steve's private key, which only Steve is privy to!

Many different symmetric and asymmetric cryptography algorithms are available, and the .NET Framework provides a number of classes to perform a number of common cryptography protocols. One of the most common (and oldest) computer cryptography protocols is *DES* (*Data Encryption Standard*), a cryptography algorithm developed by IBM in the 70s. DES is a symmetric cryptography algorithm and is supported in the .NET Framework with the `DES` class.

**NOTE**

The .NET Framework also contains a `TripleDES` class, which is an improved version of the old DES standard. `TripleDES` takes longer to encrypt and decrypt information, but is much more secure than DES.

The next two code samples, Listing 2.32 and Listing 2.33, illustrate how to both encrypt and decrypt information using the DES protocol. In Listing 2.32, the user can enter a password (which is converted into a key) and a plain-text message; these two user-supplied inputs are then plugged in to the DES algorithm and the encrypted text is outputted for the user. This encrypted output can then be taken to Listing 2.33, where the user can enter the encrypted text and a password. If the user enters the same password as she did in Listing 2.32, the encrypted text will be decrypted and the original plain-text message will be displayed.

**LISTING 2.32** A Plain-Text Message Is Encrypted with the DES Algorithm Using a User-Specified Password

```
1: <%@ Import Namespace="System.Security.Cryptography" %>
2: <%@ Import Namespace="System.Text" %>
3: <script language="VB" runat="server">
4:     Dim aConstantIV() as Byte = _
5:         { &Haa, &Hbb, &Hcc, &Hdd, &Hee, &Hff, &H12, &H78 }
6:
7:     Sub btnSubmit_OnClick(source as Object, e as EventArgs)
8:         Dim strEncryptedText as String = DESEncrypt(txtContents.Text)
9:         lblResults.Text = "<b>Encrypted String:</b><xmp>" & _
10:            strEncryptedText & "</xmp><p><i>" & _
11:            "Copy this encrypted information to the clipboard! To decrypt "
12:            "it visit <a href=""Listing2.11.2.aspx"">Listing2.11.2.aspx</a>"
13:     End Sub
14:
15:     Function GetKey(strPassword as String) as Byte()
16:         'Ensure that the strPassword string is at least 8 characters long
17:         strPassword = strPassword.PadRight(9)
18:
19:         'Now, convert the string into a byte array
20:         Dim objUTF8 as New UTF8Encoding()
21:         GetKey = objUTF8.GetBytes(strPassword.Substring(1, 8))
22:     End Function
23:
24:     Function DESEncrypt(strEncrypt as String) as String
25:         Dim aKey() as Byte
26:         Dim objUTF8 as New UTF8Encoding()
27:
28:         'Convert the password into a byte array
29:         aKey = GetKey(txtPassword.Text)
30:
31:         'Create an instance of the DES class
32:         Dim objDES as DES = DES.Create()
33:         objDES.Key = aKey
```

**LISTING 2.32** Continued

```
34:     objDES.IV = aConstantIV
35:
36:     'Convert the string into an array of bytes
37:     Dim aByteData() as Byte = objUTF8.GetBytes(strEncrypt)
38:
39:     Dim objStreamEnc as SymmetricStreamEncryptor = objDES.CreateEncryptor()
40:     Dim objCryptoStream as New CryptoMemoryStream()
41:
42:     objStreamEnc.SetSink(objCryptoStream)
43:     objStreamEnc.Write(aByteData)
44:     objStreamEnc.CloseStream()
45:
46:     'Represent the byte array as a string
47:     Dim i as Integer
48:     Dim strTmp as String = ""
49:     For i = 0 to objCryptoStream.Data.Length - 1
50:         strTmp &= objCryptoStream.Data(i).ToString() & " "
51:     Next i
52:
53:     DESEncrypt = strTmp
54: End Function
55: </script>
56:
57: <html>
58: <body>
59:   <form runat="server">
60:     <h1>Encrypt Information!</h1>
61:     <b>Enter a Password:</b>
62:     <asp:textbox id="txtPassword" TextMode="Password" runat="server" />
63:     <br><b>Enter text to encrypt:</b><br>
64:     <asp:textbox id="txtContents" TextMode="MultiLine" runat="server"
65:       Columns="50" Rows="6" />
66:     <br><asp:button id="btnSubmit" runat="server" Text="Encrypt!"
67:       OnClick="btnSubmit_OnClick" />
68:     <p>
69:     <asp:label id="lblResults" runat="server" />
70:   </form>
71: </body>
72: </html>
```

Recall that cryptography algorithms generally require two inputs: a key and the information to be encrypted. The DES algorithm (as with many other symmetric algorithms) also require a third input, an *initialization vector (IV)* that is used to get the encryption and decryption



process started. This information, as well as the key, must be in the form of a Byte array. A constant IV is defined in Listing 2.32 on lines 4 and 5.

Because there is no Page\_Load event handler, let's first turn our attention to the HTML section spanning lines 57 through 72. A post-back form is created on line 59, followed by a text box for the user to enter a password (line 62). Next, on line 64, a multiline text box is created for the plain-text message. The Submit button is created on line 66, and a label control (`lblResults`) that will display the encrypted content can be found on line 69.

After the user enters a password and encrypted text and clicks the Submit button, the form is posted back and the `btnSubmit_OnClick` event handler fires. This event handler begins on line 7 and immediately creates a String variable to hold the encrypted text, `strEncryptedText` (line 8). The encrypted information is then outputted to the `lblResults` label control (lines 9 through 12).

The `DESEncrypt` function, spanning from lines 24 through 54, does the actual work of encrypting the user-entered text. The function starts out by creating a Byte array for the key (line 25) and creating an instance of the `UTF8Encoding` class (line 26). This class is found in the `System.Text` namespace (which we imported on line 2) and is useful for quickly converting strings into byte arrays and vice versa.

Next, the `aKey` Byte array is assigned to the Byte array returned by the `GetKey` function. The `GetKey` function expects a single parameter, the user-entered password string, and converts the string into a properly-sized byte array. The DES algorithm requires a 64-bit key. Therefore, we only want the first eight characters of the user-entered password. If the password is less than eight characters, it is essential that we make it at least eight characters long. In the `GetKey` function, we ensure that the password is at least eight characters long by calling the `PadRight` method of the `String` class (line 17). Next, we create a `UTF8Encoding` instance and use the `GetBytes` method to turn the first eight characters of the password string into the byte array that is returned (lines 20 and 21).

The next step in the `DESEncrypt` function is the creation of the `DES` class instance. This is accomplished on line 32; note that this is accomplished via the `Create` static method rather than a constructor. Next, on lines 33 and 34, the `Key` and `IV` properties are set to the `aKey` and `aConstantIV` Byte arrays, respectively. Next, on line 37, the plain-text string to be encrypted, `strEncrypt`, is converted into a byte array via the `GetBytes` method of the `UTF8Encoding` class.

To actually encrypt information, a number of helper classes are required. With the DES algorithm, a `SymmetricStreamEncryptor` class is needed to perform all the dirty encryption work. A `CryptoMemoryStream` is also needed to serve as a receptacle to hold the encrypted data. On line 29, a `SymmetricStreamEncryptor` class is created and assigned to the instance returned by the `CreateEncryptor()` method of the `DES` class. Next, on line 40, a new `CryptoMemoryStream`

object is created. On line 42, `objCryptoStream`, our `CryptoMemoryStream` object instance, is set as the sink for `objStreamEnc`. This causes the encrypted information to be stuffed into `objCryptoStream`.

Next, on line 43, the plain-text message, in Byte array form, is passed through the `SymmetricStreamEncryptor` instance. The `CloseStream()` method on line 44 ends our encryption process and writes the encrypted information to the `objCryptoStream` instance.

At this point, our encrypted information is sitting in the `objCryptoStream` instance's `Data` property as a Byte array. We could use the `GetString()` method of the `UTF8Encoding` class to convert the Byte array into a string, but because not all Byte values are easily displayed through a Web browser, such a string would likely be unreadable. Therefore, we'll display each value of the Byte array numerically in a string, each value separated by a space. Lines 49 through 51 build up such a string.

Although simply being able to encrypt information is definitely neat, it is far from practical. To complete our exercise, it is essential that we create a page that allows users to enter both a password and an encrypted message and receive the decrypted, plain-text version of that encrypted message. Listing 2.33 contains the code for such an ASP.NET page.

---

**LISTING 2.33** Decrypting the Information Is as Simple as Entering a Password and an Encrypted Message

---

```
1: <%@ Import Namespace="System.Security.Cryptography" %>
2: <%@ Import Namespace="System.Text" %>
3: <script language="VB" runat="server">
4:     Dim aConstantIV() as Byte = _
5:         { &Haa, &Hbb, &Hcc, &Hdd, &Hee, &Hff, &H12, &H78 }
6:
7:     Sub btnSubmit_OnClick(source as Object, e as EventArgs)
8:         Dim strDecryptedText as String = DESDecrypt(txtContents.Text)
9:         lblResults.Text = "<b>Decrypted String:</b><xmp>" & _
10:            strDecryptedText & "</xmp>"
11:     End Sub
12:
13:     Function GetKey(strPassword as String) as Byte()
14:         'Ensure that the strPassword string is at least 8 characters long
15:         strPassword = strPassword.PadRight(9)
16:
17:         'Now, convert the string into a byte array
18:         Dim objUTF8 as New UTF8Encoding()
19:         GetKey = objUTF8.GetBytes(strPassword.Substring(1, 8))
20:     End Function
21:
22:     Function DESDecrypt(strDecrypt as String) as String
```

**LISTING 2.33** Continued

```
23: Dim aKey() as Byte
24: Dim objUTF8 as New UTF8Encoding()
25:
26: 'Convert the password into a byte array
27: aKey = GetKey(txtPassword.Text)
28:
29: 'Create an instance of the DES class
30: Dim objDES as DES = DES.Create()
31: objDES.Key = aKey
32: objDES.IV = aConstantIV
33:
34: 'Convert the string into an array of bytes
35: Dim i as Integer
36: strDecrypt = strDecrypt.Trim()
37: Dim aStringBits() as String = strDecrypt.split(" ")
38: Dim aByteData(aStringBits.Length) as Byte
39: For i = 0 to aStringBits.Length - 1
40:     aByteData(i) = aStringBits(i).ToByte()
41: Next i
42:
43: try
44:     Dim objStreamDec as SymmetricStreamDecryptor =
objDES.CreateDecryptor()
45:     Dim objCryptoStream as New CryptoMemoryStream()
46:
47:     objStreamDec.SetSink(objCryptoStream)
48:     objStreamDec.Write(aByteData)
49:     objStreamDec.CloseStream()
50:
51:     'Represent the byte array as a string
52:     DESDecrypt = objUTF8.GetString(objCryptoStream.Data)
53: catch e as Exception
54:     DESDecrypt = "Invalid password entered!"
55: end try
56: End Function
57: </script>
58:
59: <html>
60: <body>
61:     <form runat="server">
62:         <h1>Encrypt Information!</h1>
63:         <b>Enter the Password:</b>
64:         <asp:textbox id="txtPassword" TextMode="Password" runat="server" />
65:         <br><b>Enter the text to decrypt:</b><br>
66:         <asp:textbox id="txtContents" TextMode="MultiLine" runat="server"
67:             Columns="50" Rows="6" />
```

**LISTING 2.33** Continued

---

```
68:     <br><asp:button id="btnSubmit" runat="server" Text="Decrypt!"
69:                               OnClick="btnSubmit_OnClick" />
70:     <p>
71:     <asp:label id="lblResults" runat="server" />
72: </form>
73: </body>
74: </html>
```

---

The code in Listing 2.33 is nearly identical to that in Listing 2.32. The only major difference is that Listing 2.33 contains a `DESDecrypt` function instead of a `DESEncrypt` function. Other than that, there are many striking similarities between the two code listings. In fact, the `GetKey` function (lines 13 through 20) and `aConstantIV` declaration (lines 4 and 5) are identical.

Even the `DESDecrypt` function is similar to the `DESEncrypt` function. A `DES` class instance is created on line 30, and its `Key` and `IV` properties are set to the value returned by the `GetKey` function and the constant initialization vector. On lines 35 through 41, the encrypted string is converted into a `Byte` array. Recall that the encrypted string consists of numerical values separated by spaces. To turn this back into a `Byte` array, the `split` method of the `String` class is used to break up the numerical values into a `String` array (line 37). A `Byte` array is then created with the appropriate size (line 38). Finally, in lines 39 through 41, a loop is used to build up the `Byte` array one byte at a time.

Next, the decryption process is started. Rather than using a `SymmetricStreamEncryptor` instance, we use a `SymmetricStreamDecryptor` instance instead (line 44). After line 49, the `objCryptoStream`'s `Data` property contains, in `Byte` array format, the decrypted data. The `UTF8Encoding` class's `GetString()` method is used to convert this into a string (line 52).

If the user enters an invalid password, a `CryptographicException` exception is thrown. The `try ... catch` block spanning from lines 43 through 55 is in place to catch such an exception. If it occurs, the catch block starting on line 53 begins executing, and an "Invalid password entered!" message is returned (line 54).