2

Introduction to Active Server Pages

Active Server Pages were introduced by Microsoft in 1996 as a downloadable feature of Internet Information Server 3.0. The concept is pretty simple: an Active Server Page allows code written in the JavaScript or VBScript languages to be embedded within the HTML tags of a Web page and executed on the Web server. There are great advantages to this, not the least of which is security. Since your code is executed on the Web server, only HTML tags are sent to the browser. The result is that the ASP code is "invisible" to the end user.

Another upside to the "server-side script" concept is that it allows things like database connections to be made from the Web server rather than from the client. Therefore, any special configurations that might need to be set up, like ODBC data sources, only have to exist on the server. Of course, before you can create an Active Server Page (ASP), you'll need to look at the software requirements.

The Setup

Before you can create an Active Server Page, you'll need a Web server that supports Active Server Pages. The most obvious choice would be Microsoft's

Internet Information Server (IIS) version 3.0 or higher. IIS is available for Windows NT 4.0 or higher as part of the Windows NT option pack, which can be downloaded from Microsoft's Web site. For the highest level of compatibility and functionality, you'll want to use the most recent version of IIS.

Another option that you might not have considered is Microsoft's Personal Web Server for Windows 9*x* and Windows ME. If you're running Windows 98, Personal Web Server can be installed by running "setup.exe" from the Windows 98 setup CD under the "\add-ons\pws\" folder. Alternatively, it can be downloaded from Microsoft's Web site as part of the Windows NT option pack. This download is the only choice for Windows 95 or Windows ME. It's important to note that Microsoft does not support running Personal Web Server under Windows ME. While Personal Web Server is not the optimal choice for a production Web server, it is a great option for developing and testing your ASP scripts.

If you're running IIS or Personal Web Server, no additional software is required to support Active Server Pages. To allow a user to access an ASP, the ability to do so must be enabled on the IIS server. This is done by selecting "Scripts" or "Execution (Including Scripts)" from the "Home Directory" tab of the Properties window for your Web site, as shown in Figure 2.1.

For other operating systems or Web servers, it gets a little tricky, but is possible. For Unix or Linux servers running the Apache Web server, you can use a bolt-on product to add ASP support. Sun Microsystems' Sun ONE Active Server Pages (formerly called Chili!Soft ASP) is one of these products. This product supports most, but not all, of the controls available in IIS. This is just one product that can add Active Server Page support to non-Microsoft Web servers. Table 2.1 has a more complete list of ASP compatibility products and the operating systems and Web servers they run on.

There are products to allow ASPs to be used on just about any Web server out there. This fact makes using ASPs that much more attractive because you aren't limited in the choice of hardware, operating system, or Web server to host your Web pages. As you can see, there are even ASP-compatibility products for the iSeries.

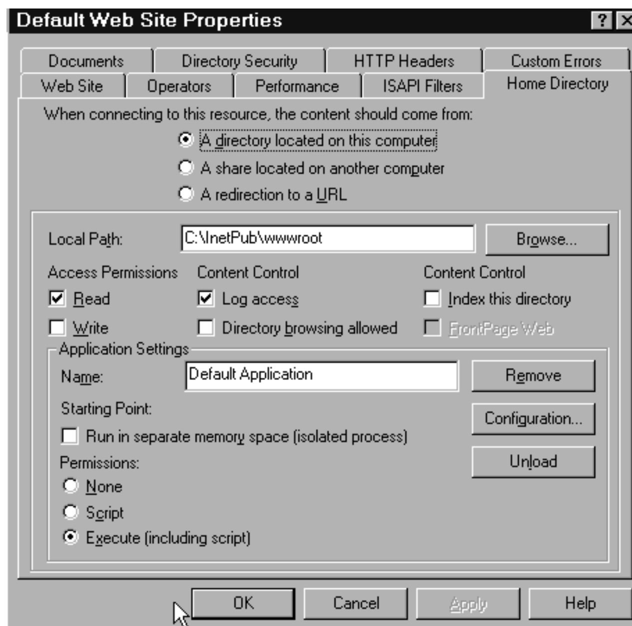

Figure 2.1: Active Server Pages are enabled by adding script permission.

Table 2.1: Compatibility Products for Using ASP on Non-IIS Web Servers

Product Name	Manufacturer	Web Servers Supported	Operating Systems
Sun ONE Active Server Page	Sun Microsystems	iPlanet, Apache, and Zeus	Solaris, Linux, Windows, and HP-UX
Instant ASP	STRYON	Apache, Oracle MS Internet Information Server (IIS), BEA Sun Web Server, WebSphere, Netscape Enterprise Server, GemStone, IBM WebSphere, Zeus, Lotus Domino WebStar	Linux, Novell Netware, Sun IBM Solaris, AIX SGI Irix, IBMi Series 400, IBM S/390 HP-UX IBM OS/2 SCO UnixWare Apple Mac OS X
Apache::ASP	Apache-ASP.org	Apache	Linux, Unix, Windows

Some Examples

Now that you've seen the requirements for using Active Server Pages, let's start examining a few basic examples. The first example uses a server-side VBScript to display a message in the browser window.

When you are creating an Active Server Page, the server-side script can be identified in one of the following two ways:

- Use the server-side script block identifiers "<%" and "%>".
- Use the <SCRIPT> tag with the RUNAT="SERVER" attribute.

Functionally, both of these give you the same result, but it's important to note that the latter is the only one allowed in ASP.NET (discussed in a later chapter). Using either of these methods, you can mix your script blocks with static HTML tags. Since chapter 1 covered HTML, I won't spend a lot of time on HTML tags here, except to review those tags that are pertinent to the examples shown.

The sample ASP code shown in Figure 2.2 will display the current time in the browser window. To try this example, enter the code and save the file to the default directory of your Web server (for example, "C:\Inetpub\wwwroot on IIS"). Assuming that this file were named "time.asp," you would access it from a Web browser by entering the following URL: *http://Webserver/time.asp*

```
<HTML>
<HEAD>
<TITLE>Sample ASP Clock</TITLE>
</HEAD>
<BODY BGCOLOR="BLACK">
<FONT COLOR="GREEN">
<SCRIPT LANGUAGE="VBScript" RUNAT="SERVER">
RESPONSE.WRITE NOW()
</SCRIPT>
</BODY>
</HTML>
```

Figure 2.2: This Active Server Page displays the current time and date.

When this page is loaded into your browser, right-click it and select “View Source.” The HTML shown in Figure 2.3 should be displayed. Notice that the script block is not visible to the user.

```
<HTML>
<HEAD>
<TITLE>Sample ASP Clock</TITLE>
</HEAD>
<BODY BGCOLOR=”BLACK”>
<FONT COLOR=”GREEN”>
03/01/2003  11:31:04
</BODY>
</HTML>
```

Figure 2.3: The viewable source does not include the script block.

ASP Objects

While the script portion of the previous example is pretty basic, it gives a good example of what Active Server Pages are all about. First, HTML defines the page title, background color, and font color. Then, the script code uses the Write method of the Response object to send output to the browser.

When writing ASP scripts, a special set of ASP objects are available within the VBScript language to assist in the programming process. These objects give you access to *application programming interfaces* (*APIs*) that allow you to manipulate the document displayed in the browser. Each of these objects has the following:

- *Properties* set or read information about the object.
- *Methods* execute an action on an object.
- *Collections* contain items related to the object.

The Response object will be used heavily throughout all of the examples in this book. This object allows you to control what is displayed in the browser. Table 2.2 shows all of the available properties, methods, and collections for the Response object.

Table 2.2: The Properties, Methods, and Collections of the Response Object

Name	Type	Values	Description
AddHeader	Method	N/A	Sets the value of a specified HTML header.
AppendToLog	Method	N/A	Adds a specified string to the end of the Web server log entry.
Buffer	Property	True/false	Defines whether or not to buffer content before sending it to the browser.
CacheControl	Property	Public/private	Defines whether or not page contents are cached on a proxy server.
Charset	Property	String value	Inserts a character-set name into the content-type header.
Clear	Method	N/A	Clears out the contents of the buffer.
ContentType	Property	MIME type	Defines the HTTP content type of data being sent to the browser.
Cookies	Collection	N/A	Used to set or read cookie values.
End	Method	N/A	Stops further script processing and sends buffer content to the browser.
Expires	Property	Numeric value	Define the length of time before a cached page expires. A value of zero disables caching of this page.
Expires Absolute	Property	Date/time	Defines the date and time when a cached page expires.
Flush	Method	N/A	Sends the contents of the buffer to the browser and clears it.
IsClient Connected	Property	True/false	Identifies whether or not the client is connected to the server.
Pics	Property	String value	Defines the PICS content rating.
Redirect	Method	N/A	Redirects control to a specified page. Only valid if no HTML headers have been sent to the browser.
Status	Property	String value	Contains the contents of the status line returned by the server.
Write	Method	N/A	Sends output to the browser.

Table 2.3: Objects Used within an Active Server Page

Object	Description
Application	Defines or reads variables that are specific to a Web application (a Web site counter, for example).
Request	Allows values to be retrieved from the client browser.
Response	Sends data to the browser window.
Server	Controls various attributes related to the Web server itself.
Session	Creates or reads values that are specific to a user's current session.

As you can see, the Response object controls output that is sent to the browser. For example, the Cookies collection, when used with the Response object, would define Cookie values that are stored on the client computer, to be retrieved by the application at a later time.

Active Server Pages have their own subset of objects used to control input from the client browser, output to the browser, and values used by the Web application.Table 2.3 contains a list of these objects and the functions they perform.

In the same way that the Response object writes information out, the Request object reads information in. One of the primary uses of Request is to read information sent into the Active Server Page. This is done using either the Querystring or Form collections. Each of these collections allows you to read "variables" from another Web page. The Querystring collection accesses variables supplied as part of the query string that is appended to the URL with a question mark:

http://myserver/myfirst.asp?fname=JOHN&lname=DOE

Query strings are generated automatically by an HTML form that uses the GET method, but they can also be manually inserted into a hyperlink. In the example link, the variables "fname" and "lname" are sent to the ASP specified. Notice that the ampersand character separates the query string variables. To read these variables into the ASP, you would use the following two lines of VBScript:

```
FirstName=REQUEST.QUERYSTRING("fname")
LastName=REQUEST.QUERSTRING("lname")
```

The one downside to using the Querystring collection is that the variable names and values can be seen within the URL on the browser. This can be a problem if you need to supply information to your Active Server Page that you don't want the user to see. This problem can be avoided by using the Form collection.

Like Querystring, values for the Form collection can be passed automatically from an HTML form. The difference is that the HTML <FORM> tag must use the POST method rather than the GET method. When the HTML form is submitted, any objects within the form will be passed through the Form collection. The Form variables are read using the same method as Querystring variables. Form variables, however cannot be appended to the URL as Querystring variables can.

In some cases, you won't always know the names of the Querystring or Form variables. To allow for this, you can use the For Each..Next loop. This loop is similar to a standard For.Next loop, with a few exceptions. A regular For.Next loop bases its looping on a starting value and an ending value. The resulting Numeric field is incremented based on the Step value provided. The For Each..Next loop feeds the name of each Item within the specified Collection into the supplied variable.

Here is an example of how to read all Form variables using a For Each..Next loop:

```
<%
For Each var In Request.Form
   Response.Write var & " = " & Request.Form (var)
Next
%>
```

In this example, each Form variable and its value will be displayed in the Web browser.

The Request object can do much more than just pass values between Web pages. Table 2.4 contains a list of the properties, methods, and collections available for the Request object.

Variable data within any of the five collections in the table can be retrieved simply by specifying *Request("variablename")*. When this form of the Request object is used, the application will search through each of these collections to find the matching variable. Since searching through each of the collections can be time-consuming, the preferred method is to retrieve data through the specific collection name. There are circumstances, however, when you might want to use this functionality. For example, your application might sometimes supply variable data through the Querystring collection, and other times use the Form collection. Using the search feature, your application could simply access the variable using the *Request("variablename")* form, so it wouldn't have to deal with figuring out which collection was used.

The Server Variables collection contains information specific to your server and the client connected to it. For example, the command below would retrieve the authorized user name of the user requesting the Web page:

Table 2.4: The Properties, Methods, and Collections of the Request Object

Name	Type	Description
Binary Read	Method	Retrieves a specified number of bytes of data supplied by the POST method into a safe array.
ClientCertificate	Collection	Retrieves the value of variables in the client certificate that is sent in the request.
Cookie	Collection	Reads the value of cookies.
Form	Collection	Reads the value of Form variables.
Querystring	Collection	Contains all variables supplied to the page through the query string.
ServerVariable	Collection	Reads attributes of the Web server or the client browser.
TotalBytes	Property	Defines the total number of data bytes sent by the client request; read-only.

```
User=Request.ServerVariable("AUTH_USER")
```

In this case, the AUTH_USER server variable is used to obtain the desired information. This specific variable will only return a value if your Web site requires a user name and password. If the Web site allows anonymous access, this variable will return an empty string. A list of some of the available ServerVariable items can be found in Table 2.5.

These values can be used to control the flow of your application in many ways. For example, you could use information from the HTTP_USER_AGENT variable to determine that a client request came from a PDA running Microsoft Pocket PC, and then redirect to a page specifically formatted for that device. The ASP code shown below would accomplish this, using the Response.Redirect method:

```
<%
UserAgent = Request.ServerVariables("HTTP_USER_AGENT")
IsPocketPC = (InStr(UserAgent, "Windows CE") > 0)

If IsPocketPC Then
   Response.Redirect("pda.asp")
Else
   Response.Redirect("default.asp")
End If
%>
```

This example first places the value of the HTTP_USER_AGENT server variable into the program variable UserAgent. Next, the InStr function looks for the string “Windows CE” within the UserAgent variable. If there is a match, the value of IsPocketPC will be TRUE; otherwise, it will be false. Finally, this value is used with an IF..THEN statement to redirect the browser to the appropriate page for the device.

The Session and Application objects perform similar functions. These objects allow you to define variables that will be available to multiple pages within your application, without having to be passed through the Querystring or Form collections. To set or read a Session variable, you reference the variable as you

Table 2.5: Commonly Used ServerVariable Items

Variable Name	Description
ALL_HTTP	Retrieves all of the HTTP headers from the client.
APPL_PHYSICAL_PATH	Retrieves the physical path corresponding to the metabase path returned in APPL_MD_PATH. The metabase is the database used internally by IIS to store Web server settings.
AUTH_PASSWORD	Returns the user password if basic authentication is used.
AUTH_TYPE	Returns the authentication method used to validate users when they attempt to access a protected script.
AUTH_USER	Supplies the user name sent in the client's authorization header.
HTTP_HOST	Contains the host name of the Web server.
HTTP_USER_AGENT	Returns a string containing information about the client. This string can be used to determine the operating system and browser on the client.
HTTPS	Indicates whether the request came on a secure port.
LOCAL_ADDR	Identifies the IP address on which a request came in.
LOGON_USER	Returns the Windows account used to access the page.
REMOTE_ADDR	Returns the IP address of the client.
REMOTE_HOST	Identifies the name of the client system.
REQUEST_METHOD	Determines the method used to send data to the server (GET or POST).
SERVER_NAME	Provides the Web server's host name.
SERVER_PORT	Returns the TCP/IP port on which a request came in.
SERVER_SOFTWARE	Returns name and version information for the Web server software.
URL	Returns the base portion of the URL page.

would any item within a collection. The statement below would read the value of the Session variable "customer" into the field Cust:

```
CUST = SESSION("customer")
```

Session variables exist until the user ends the current session with the Web server, or until Session.Abandon is executed. This method removes any session variables for the current session. In addition to the Abandon method, the Session object supports two properties, SessionID and Timeout. The SessionID property contains a unique numeric identifier for the current session. The Timeout property specifies a timeout value for the current session.

You would use the Session object in the same way as the LDA or QTEMP from within an application on the iSeries. Just like the LDA, Session variables are only available to the session running the application. Like objects stored in QTEMP, objects created using the Session object no longer exist when the session is closed.

The Application object exists from the time the Web application is created. A Web application is defined as a set of Web pages under a common root folder. Application variables are cleared when the Web server is restarted. Application variables are defined by the same method as Session variables. The line below shows how an Application variable would be defined:

```
Counter=Application("Counter")
```

The value of the Application variable is available to all Sessions running on the Web server. For this reason, Application variables are an ideal way to create a page counter. You use Application objects in the same way that you use data areas on the iSeries. These variables can be used to store values and share them between applications.

The Server object is used to access methods and properties of the Web server. This object performs several functions, not the least of which is to provide access to an ActiveX control on the server. Table 2.4 lists the properties and methods of the Server object.

The ASPError object is used in conjunction with the Server.GetLastError method. This object retrieves information about the most recent ASP error that has occurred. This object supports the properties listed in Table 2.5.

Table 2.4: The Properties and Methods of the Server ASP Object

Name	Type	Description
ScriptTimeout	Property	Defines the amount of time the ASP script runs before a timeout occurs.
CreateObject	Method	Creates a server-side ActiveX control.
Execute	Method	Calls another ASP page as though it were part of this page.
GetLastError	Method	Builds an ASPError object containing details of the most recent error.
HTMLEncode	Method	Replaces any special characters in a specified string with their HTML-encoded equivalents.
MapPath	Method	Returns the physical disk path for the provided virtual path.
Transfer	Method	Transfers control to a specified ASP page while leaving any ASP objects intact.
URLEncode	Method	Encodes a specified URL in the same way that HTMLEncode encodes a text string.

Table 2.5: The Properties of the ASPError Object

Property	Description
ASPCode	Returns the ASP error code related to the error.
Number	Results in the numeric error code returned by a COM component.
Source	Supplies the actual source code for the line in error.
Category	Provides a string value identifying what generated the error (IIS, a COM component, or a scripting language).
File	Contains the name of the ASP file where the error originated.
Line	Returns the line number in error.
Column	Identifies the column position of the error within the error line.
Description	Contains a short text description of the error.
ASPDescription	For ASP errors, provides a longer description of the error.

To access the ASPError object, you first need to define the relationship to the Server object, as shown here:

```
Set objASPError = Server.GetLastError
```

In this example, all of the ASPError object properties are now available to the object objASPError. This object can be used to create custom error-trapping for your application. All errors that are generated access a default Web page on IIS named “500-100.asp.” This file can be overridden to a specified custom ASP file that uses the ASPError object. To override, follow these steps:

1. From the IIS Administrator, right-click on your Web site.
2. Select Properties, then select Customer Errors, and look for 500;100.
3. Click Properties, then click Select URL.
4. Enter the URL for the modified version of the “500-100.asp” page.

Within this modified page, you might choose to write error entries to a database file that contains an error log. You might also send an e-mail message to an administrator to notify him or her that an error has occurred with the application. You’ll see how to accomplish each of these a little later on.

The final ASP object that we’ll examine here is the ObjectContext object. ObjectContext allows you to create transactional ASP scripts. Creating ASP scripts that use transaction processing allows you to commit or abort a group of transactions performed on an object in a single step. You would use ObjectContext in the same way that SQL uses transaction processing. This is also similar to using commitment control in an iSeries application.

ObjectContext has only two methods. The ObjectContext.SetComplete method finalizes all transactions performed within the script, while ObjectContext.Abort cancels any transactions performed within the script. For transaction processing to be active, the script must contain the @TRANSACTION directive as its first line, as shown here:

```
<@TRANSACTION value>
```

The value specified is used to define whether or not transaction processing will be used. A value of either REQUIRED or REQUIRED_NEW will cause a transaction to be initiated. If values of SUPPORTED or NOT_SUPPORTED are used, a transaction will not be initiated. One transaction can span multiple ASP pages if control is transferred through the use of the Server.Transfer or Server.Execute method. In either of these cases, if REQUIRED is specified on the @TRANSACTION directive, the new page will continue within the transaction of the original ASP script. If the original script did not use transaction processing, the new script will create a new transaction.
The code snippet in Figure 2.4 shows an example of how ObjectContext is used.

```
< @TRANSACTION REQUIRED >
```

In this example, the SetComplete method is used. This is not necessary, however, since the transaction will automatically be set complete upon completion of the script.

```
<HTML>
<BODY>
<SCRIPT LANGUAGE="VBScript" RUNAT="SERVER">
Set objMine=Server.CreateObject("MyObj.Name")
If objMine.Value=0 Then
   ObjectContext.SetAbort
Else
   ObjectContext.SetComplete
End If
</SCRIPT>
</BODY>
</HTML>
```

Figure 2.4: This script uses ObjectContext for transaction processing.

Using HTML Forms with ASP

Now that you've seen each of the ASP objects, let's examine how to use these objects to create Web pages. ASP scripts work together with HTML in an Active Server Page in the same way that RPG works with a display file to create an interactive application on the iSeries. As you saw earlier, the Response.Write method can be used to send output to the browser. This output can include HTML tags, which means that your application can dynamically create the page.

To create input fields within Web pages, use HTML forms. The values entered into the HTML form are passed to the ASP script through the Request.Querystring and Request.Form collections. Figure 2.5 contains code for a simple HTML form.

This sample first uses the <FORM> tag to define form information. The NAME attribute identifies the form, while the METHOD attribute defines how the values from the form will be passed. A method of POST sends the variables through the Request.Form collection. The GET method will send the values through the Request.Querystring collection.

Next, in addition to some simple headings, each of the input fields is defined. Again, the NAME property is used to identify the variable within the Request.Querystring or Request.Form collection. The TYPE property defines the type of input field being created. All of the fields here are simple text boxes. The SIZE property defines the size of the text box as a number of characters. The

```
<html>
<body>
<form name="sampleform" method="POST" action="process.asp" ID="Form1">
Name: <input type="TEXT" name="Name" size =35 ID=""text"1"><br>
Address:<input type="TEXT" name="Addr1" size=35 ID=""text"2"><br>
City, St ,Zip: <input type="TEXT" name="City" size=15 ID=""text"3"> 
<input type="TEXT" name="State" size=2 maxlength="2" ID=""text"4"> 
<input type="TEXT" name="Zip" size=5 maxlength="5" ID=""text"5"><br>
<input type="SUBMIT" value="Process" ID=""submit1" NAME="submit1">
</form>
</body>
</html>
```

Figure 2.5: This HTML code creates an input form.

MAXLENGTH property used on some of the fields defines a maximum accepted length for the variable.

The final field used within the form is also an input field, but this time it is defined as SUBMIT, which creates an input button that can be used to send the form to the server. The NAME parameter is used once again to identify this item when the form is submitted. The VALUE property, in this case, serves two purposes. First, it provides the text that will be displayed on the button. Second, it is the value used for the SUBMIT button when the form is submitted.

Using a text editor, enter the code in Figure 2.5 and save it with the name “sample.html.” Now, open the file in your Web browser by typing the full path to the file, including the file name, in the address bar. You should see something similar to Figure 2.6.

At this point, you might be wondering where the ASP code comes in. Notice that the ACTION parameter on the <FORM> tag defines a page called “process.asp.” This is the Active Server Page that will process this form. Figure 2.7 contains the code for this page.

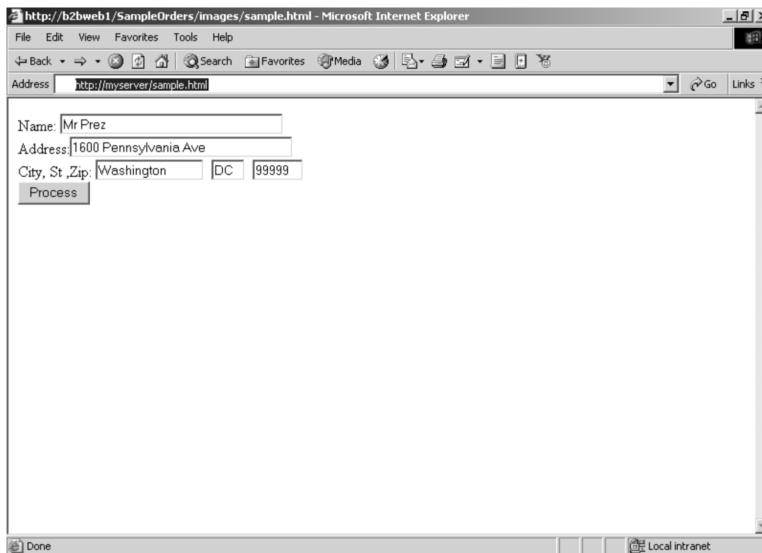

Figure 2.6: This is how the sample form in Figure 2.5 will look in the browser.

```
<html>
<body>
<script language="VBScript" runat="Server">
Dim var, val
For Each X in Request.Form
   var=X
   val=Request.Form(X)
   Response.Write var & " = " & val & "<br>"
Next
</script>
</body>
</html>
```

Figure 2.7: This ASP script displays the data from the form in Figure 2.6.

This example will take the values entered on the sample HTML form and display the variable names and their values in the browser window. The For Each..Next statement reads all of the values within the Request.Form collection. The values are sent out to the browser window using the Response.Write method. Notice that the HTML tag <BR> inserts a line break at the end of each line. This is a perfect example of how to combine variable values with HTML tags to create dynamic output. Figure 2.8 shows how the formatted output will appear in the browser window.

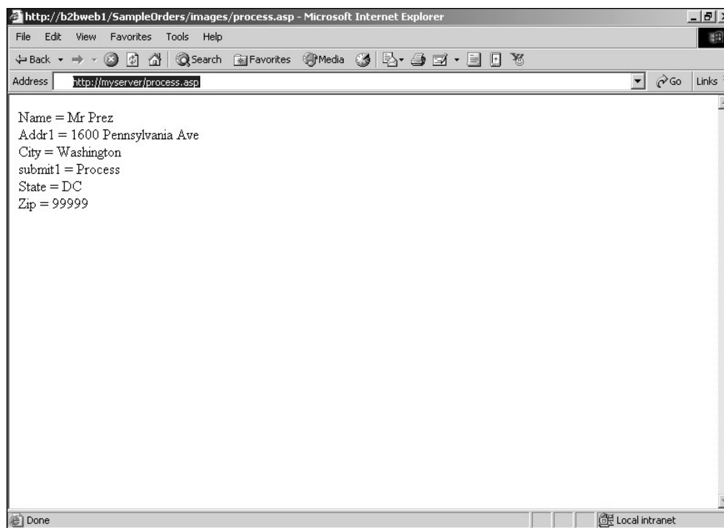

Figure 2.8: This is the ASP output from the script in Figure 2.7.

To take this example a little further, you can use an Active Server Page to create the HTML form. This gives you the ability to include or exclude form items dynamically. The example in Figure 2.9 creates values in a list box using a For..Next loop.

In this example, the <SELECT> tag creates a list box. The <OPTION> tags that provide the content for the list box are created by ASP code. The text between <OPTION> and </OPTION> defines what will be displayed for each option's value. This would allow you, for example, to display state names within a list, but return the state abbreviation to your ACTION page. Figure 2.10 shows what the output for this page would look like in a browser window. Using the same "process.asp" page created earlier, the value selected will be displayed in the browser window.

In both of these examples, the <INPUT> tag with TYPE of "SUBMIT" is used to create a command button to send the screen back to the server for processing. It's pretty easy to draw a correlation between this button and a function key within a display file on the iSeries. When you also consider that it's possible to have multiple submit buttons within the same form, it's even easier to see how similar these buttons are to function keys.

```
<html>
<body>
Select a range of values:
<form name="sampleform" method="POST" action="process.asp">
<select name="listvalue">
<script language="VBScript" runat="Server">
For I = 0 to 150000 step 15000
   J=I+14999
   Response.Write "<option value='" & I & "'>" & I & " - " &  _
   J & "</option>"
Next
</script>
</select>
<input type=SUBMIT value="Submit Page">
</form>
</body>
</html>
```

Figure 2.9: This HTML code creates an input form.

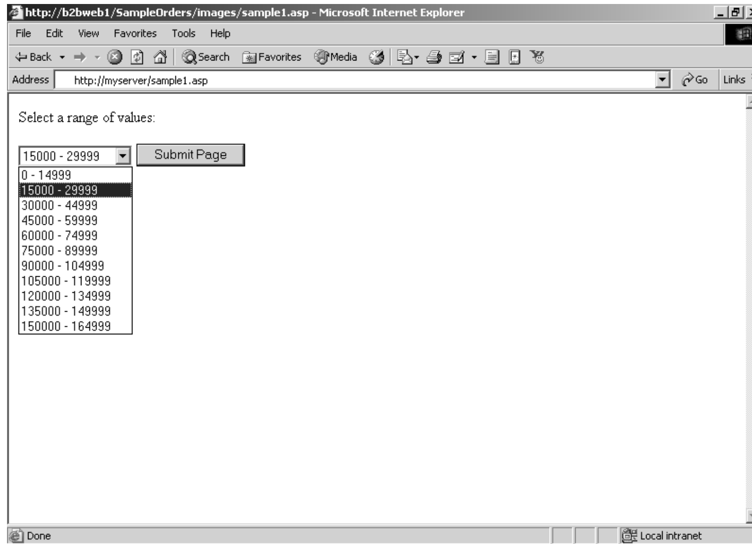

Figure 2.10: The options in this list box are created by an ASP page.

The example in Figure 2.11 takes this idea even further, by creating an HTML form that displays three "functions" along the bottom of the form. The value returned into the input field named "Function" determines how the page defined on the form's ACTION parameter should proceed. This is done similarly to the way an RPG program might deal with function keys.

```
<html>
<body>
<form name="sampleform" method="POST" action="process2.asp">
Name:<input type="TEXT" name="Name" size =35><br>
Address:<input type="TEXT" name="Addr1" size=35><br>
City, St, Zip:<input type="TEXT" name="City" size=15> 
<input type="TEXT" name="State" size=2 maxlength="2"> 
<input type="TEXT" name="Zip" size=5 maxlength="5"><br>
<input type="SUBMIT" name="function" value="Update">
<input type="SUBMIT" name="function" value="Cancel">
<input type="SUBMIT" name="function" value="Display">
</form>
</body>
</html>
```

Figure 2.11: This sample uses command buttons in place of function keys.

The ASP code for “process2.asp” is shown in Figure 2.12. This sample first reads the value of the Function variable, which contains the defined value for the SUBMIT button. Next, the Select Case statement controls the program flow based on each of the possible “function key” values. The basic program logic used in Figure 2.11 is very similar to how you would accomplish the same task in an RPG program.

In addition to simple text fields and command buttons, you can also create radio buttons and check boxes, and send their values through the Form or Querystring collections. Radio buttons are created by using the <INPUT> tag with

```
<!-- include functions.asp -->
<html>
<body>
<script language="VBScript" runat="Server">
Dim Fnc

' Read "function" variable from the FORM collection
Fnc=Request.Form("function")

' Use SELECT CASE to control the application flow
Select Case Fnc
   ' Call Update Routine
      Case "Update"
         Call Update()
   ' Redisplay main screen
      Case "Cancel"
         Response.Redirect "default.html"
   ' Display Form collection
      Case "Display"
         For Each x In Request.Form
            VarName=x
            VarVal=Request.Form(x)
         If VarName<>"function" Then
            Response.Write VarName * " = " & VarVal
         End If
         Next
End Select
</script>
</body>
</html>
```

Figure 2.12: This sample uses command buttons in place of function keys.

TYPE="RADIO." Radio buttons are normally used for multiple-choice selections. You define multiple radio buttons as one group by giving them the same name, as shown in Figure 2.13.

When this example is loaded into your browser, you'll see that by selecting one option, any other that is selected is unselected automatically. When the page is submitted, the value of the variable "Send" will reflect the option that was highlighted. Use a check box, on the other hand, when you want to allow an option to be defined as "on" or "off." If you were to change the TYPE on the example in Figure 2.13 from RADIO to CHECKBOX, it would be possible to check all three options at the same time. When the page was submitted, all three values would be returned into the Form collection. Table 2.6 contains a complete list of the Form objects available.

While ASP isn't a requirement to use HTML forms, HTML forms are a key part of Active Server Pages. This is because you use HTML forms to pass values to the ASP scripts. There are times, however, when you'll want to do some type of validation prior to sending the form back to the server. For this, you can use *client-side scripts*.

Client VBScripting and JavaScripting

In addition to creating scripts that run on the server, you can also create scripts in VBScript or JavaScript that run within the client browser. This ability is

```
<html>
<body>
<form name="sendto" method="POST" action="report.asp" ID="Form1">
Send Report To:<br><input type="RADIO" name="send"
value='P'>Printer</input><br>
<input type="RADIO" name="send" value='F'>File</input><br>
<input type="RADIO" name="send" value='S'>Screen</input><br>
<input type="SUBMIT" value="Process" ID=""submit1" NAME="submit1">
</form>
</body>
</html>
```

Figure 2.13: Radio buttons are created using the <INPUT> tag, as shown here.

Table 2.6: Objects That Can Be Used within an HTML Form

HTML Tag	Description
<INPUT TYPE="BUTTON">	Displays a command button on the form. This type of button does not automatically cause any action on the form.
<INPUT TYPE="CHECKBOX">	Appears on the form as a check box.
<INPUT TYPE="IMAGE">	Displays a specified image file on the form that will submit the form when clicked.
<INPUT TYPE="PASSWORD">	Displays a text box in which all entered characters appear as asterisks. This type is used for password entry.
<INPUT TYPE="RADIO">	Displays a radio button within the form.
<INPUT TYPE="RESET">	Appears as a command button, but will reset all of the values on the form to their defaults.
<INPUT TYPE="SUBMIT">	Displays a command button on the form that will automatically submit the form to the server.
<INPUT TYPE="TEXT">	Provides a single-line text box within the form.
<SELECT> <OPTION>	Act together to create a list box on the form. The SELECT tag defines the list box itself. The OPTION tag defines each of the items within the list box.
<TEXTAREA>	Similar to the TEXT type, but allows multiple lines of text to be entered into one field.

a double-edged sword, however. It does allow you to read the contents of a form prior to sending it to the server. At the same time, client scripts must be written in a way that allow for maximum compatibility. Different browsers might support different levels of client scripting. It's also entirely possible that a client might not have client scripts enabled at all. In any case, the following examples provide an introduction to client-script techniques.

Handling Mouse Events

Many of the objects within a Web page have events that can have client-scripting functions assigned to them. For example, a cell within an HTML table has ONMOUSEOVER and ONMOUSEOUT events. These two events are fired

when the pointer is moved over or off of the cell, respectively. You can use these events to change the appearance of the cell when the user moves the pointer over it.

The Web page in Figure 2.14 uses a client script to highlight the text when the mouse pointer is placed over each of the cells. This example creates three cells within a table, and highlights each cell as the mouse pointer moves over it.

```
<html>
<title>Mouse Over Client Script Example</title>
<body>
<table border=1>
   <tr>
      <td onmouseover='mouseover()' onmouseout='mouseout()'>
         Cell 1
      </td>
      <td onmouseover='mouseover()' onmouseout='mouseout()'>
         Cell 2
      </td>
      <td onmouseover='mouseover()' onmouseout='mouseout()'>
         Cell 3
      </td>
   </tr>
</table>
</body>
<script language='JavaScript'>
function mouseover(){
 var srcElement;
 srcElement=window.event.srcElement;
 srcElement.style.color= "White";
 srcElement.bgColor = "Black";
}

function mouseout(){
 var srcElement;
 srcElement=window.event.srcElement;
 srcElement.style.color= "Black";
 srcElement.bgColor = "White";
}
</script>
</html>
```

Figure 2.14: This sample assigns a client-side JavaScript to an event.

In this example, you determine the cell that is highlighted by using the Window.Event.srcElement object. The Window object is used to read and set attributes of the browser window or items within the browser window (in this case, the cells). After assigning the Window.Event.srcElement object to a variable named srcElement, you can redefine properties of the cell. In this case, the code changes the background color using the bgColor property, and changes the font color using the Style.Color property.

Figure 2.15 shows what this code would look like in a browser. In the browser, the cell coloring would change when the mouse pointer hovered over it. The coloring would return to normal when the ONMOUSEOUT event fired for the cell.

Client-side Form Validation

You can use this same technique to validate an entry in a form prior to submitting the form to the server. This is done by creating a function for use with the form's ONSUBMIT event. This event is fired when the form is submitted

Figure 2.15: The "Mouseover" example as it appears in the browser.

through the use of a SUBMIT button or image. This technique can be used for basic data validation, like checking for the length of data entered into a field or verifying that a given field has an entry. The example shown in Figure 2.16 uses a client-side Java Script to validate three different form fields.

Each of the fields in Figure 2.16 has its own validation check. The Name field is simply checked to ensure that a value was entered. This is done through the use of a JavaScript “If” statement.

```
<html>
<body>
<form name="form1" method="POST" action="p2.asp"
onsubmit="return validate();">
* = Required Field<br>
Name *: <input type="TEXT" name="FullName" size =35 ID="FullName"><br>
E-Mail:<input type="TEXT" name="EMail" size=35 ID="EMail"><br>
Phone #:<input type="TEXT" name="Phone" size=15 ID="Phone"><br>
<input type="SUBMIT" value="Submit" ID="submit1" NAME="submit1">
</form>
</body>
<script language="JavaScript">
function validate() {
   if (form1.FullName.value == '') {
      window.alert('A Name is Required');
      form1.FullName.focus();
      return(false);
   }
   if (form1.EMail.value.search('@')==-1) {
      window.alert('E-mail address is invalid');
      form1.EMail.focus();
      return(false);
   }
   if (form1.Phone.value.length<10) {
      window.alert("Phone number is invalid");
      form1.Phone.focus();
      return(false);
   }
   return true;
}
</script>
</html>
```

Figure 2.16: This example uses a client-side script for data validation.

In JavaScript, all grouped statements (If, For, etc.), along with functions themselves, are grouped using the { and } bracket characters. For functions, this is done using the following format:

```
function  functioname() {
' function code
}
```

Rather than require an End Function statement as would be used in a VBScript, you simply use the close bracket. An If statement is defined in a similar manner:

```
if (condition) {
  '  conditional code
}
```

This example shows that the If statement is followed immediately by the condition enclosed in parentheses, and then the conditional code enclosed in brackets. Each of the fields within a form is referenced as shown here:

```
formname.fieldname.value
```

In this example *formname* corresponds to the name of an HTML form. *Fieldname* references the input field within that form. The *value* property retrieves the value of the field.

To get back to the example in Figure 2.16, the first If statement simply checks to see whether FullName is blank. If this field is blank, a message box signifying the error will be displayed using the Window.Alert command. Next, the Focus() method is used on the FullName field in the same way that a Position Cursor Display Attribute [DSPATR(PC)] would be used in a display file on the iSeries. It will cause the cursor to be placed in the FullName field. Then,

the return(False); command is used. This will cause the HTML form to be redisplayed, after displaying the message box.

The Email field's address validation is slightly more complex. For this field, the script not only ensures that the field not contain a value, but specifically that it contains the @ character. This is done by using the Search JavaScript function. This function scans the string on which it is used for the specified string value. If the value is not found, a message box is displayed, notifying the user of the error. As with the FullName, the Focus() function positions the cursor to this field when the form is redisplayed by returning the False value.

The third validation is on the Phone field. This validation checks to ensure that the value in this field is no less than 10 positions. If the value is less than 10 positions, the same logic used for the other fields is used to display a message box and redisplay the form, with the cursor positioned at this field. Figure 2.17 shows an example of what the output from this sample page will look like.

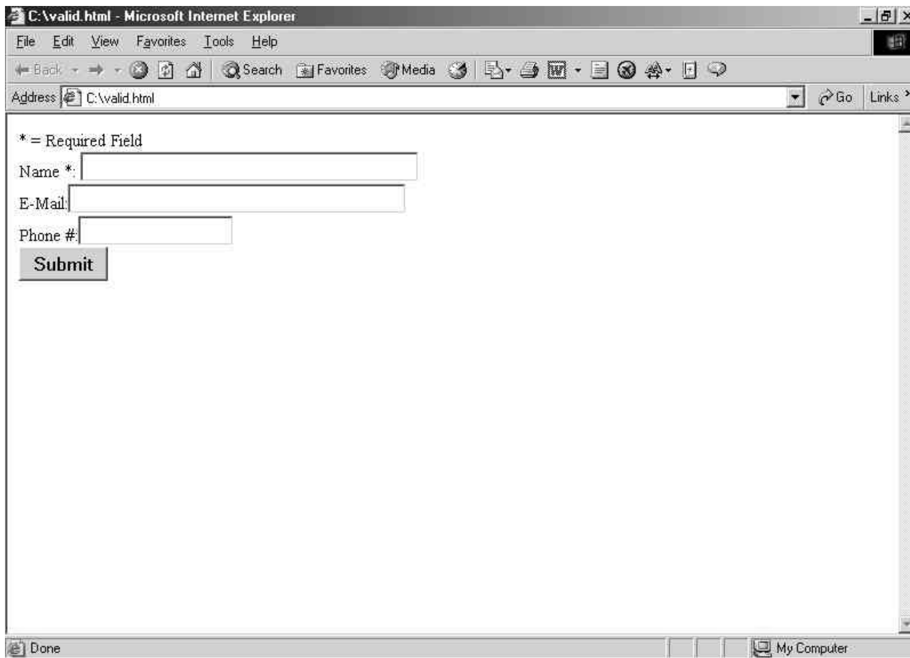

Figure 2.17: Each of these fields is validated using a client-side JavaScript.

VBScript Versus JavaScript

The two examples you just looked at both use JavaScript code running on the client. For these examples to work properly, the browser must support JavaScript. Another option for client-side scripting is to use the Visual Basic scripting language. This is basically the sample scripting language you use with server-side scripting, minus any of the ASP objects. The code in Figure 2.18 contains a sample client VBScript.

This example is basically a duplicate of Figure 2.14, with one exception: the JavaScript code has been replaced with its VBScript equivalent. The similarities

```
<html>
<title>Mouse Over Client VBScript Example</title>
<body>
<table border=1>
   <tr>
      <td onmouseover='mouseover' onmouseout='mouseout'
      language=VBScript> Cell 1 </td>
      <td onmouseover='mouseover' onmouseout='mouseout'
      language=VBScript> Cell 2 </td>
      <td onmouseover='mouseover' onmouseout='mouseout'
      language=VBScript> Cell 3 </td>
   </tr>
</table>
</body>
<script language='VBScript'>
sub mouseover()
 Dim obj
 Set obj=window.event.srcElement
 obj.style.color="Red"
 obj.bgColor="Black"
end sub

sub mouseout()
 Dim obj
 Set obj=window.event.srcElement
 obj.style.color="Black"
 obj.bgColor="White"
end sub
</script>
</html>
```

Figure 2.18: This example uses a VBScript version of the Mouseover and Mouseout events.

include the use of the window.event.srcElement object. The differences including the use of the Dim statement in place of the Var statement to define variables. Functionally, this example is identical to Figure 2.14.

The decision to use either JavaScript or VBScript involves several factors, including your comfort level with each of these languages. In the end, however, the deciding factor might be browser compatibility. As a general rule, JavaScript is supported by more browsers that VBScript, so in my opinion JavaScript would be the client-side language of choice.

Summary

As you've seen in this chapter, ASP scripts allow you to dynamically adjust the HTML within a Web page. The ability to execute scripts on the Web server lets you control the output sent to the browser, while keeping these scripts "hidden" from the end user. In chapter 3, we'll take this to the next level by examining how to access data stored in the iSeries (or any other database) from within an ASP.