Active Server Pages

Introduction to Active Server Pages

Microsoft® Active Server Pages (ASP) is a *server-side scripting* environment that you can use to create and run dynamic, interactive Web server applications. With ASP, you can combine HTML pages, script commands, and COM components to create interactive Web pages or powerful Web-based applications, which are easy to develop and modify.

For the HTML Author

If you are an HTML author, you will find that server-side scripts written in ASP are an easy way to begin creating more complex, real-world Web applications. If you have ever wanted to store HTML form information in a database, personalize Web sites according to visitor preferences, or use different HTML features based on the browser, you will find that ASP provides a compelling solution. For example, previously, to process user input on the Web server you would have had to learn a language such as Perl or C to build a conventional Common Gateway Interface (CGI) application. With ASP, however, you can collect HTML form information and pass it to a database using simple server-side scripts embedded directly in your HTML documents. If you are already familiar with scripting languages such as Microsoft VBScript or Microsoft® JScript® (JScript is the Microsoft implementation of the ECMA 262 language specification), you will have little trouble learning ASP.

For the Experienced Web Scripter

Since ASP is designed to be language-neutral, if you are skilled at a scripting language such as VBScript, JScript, or PERL, you already know how to use Active Server Pages. What more, in your ASP pages you can use any scripting language for which you have installed a COM compliant scripting engine. ASP comes with VBScript and JScript scripting engines, but you can also install scripting engines for PERL, REXX, and Python, which are available through third-party vendors.

For the Web Developer and Programmer

If you develop back-end Web applications in a programming language, such as Visual Basic, C++, or Java, you will find ASP a flexible way to quickly create Web applications. Besides adding scripts to create an engaging HTML interface for your application, you can build your own COM components. You can encapsulate your application's business logic into reusable modules that you can call from a script, from another component, or from another program.

The Active Server Pages Model

A server-side script begins to run when a browser requests an .asp file from your Web server. Your Web server then calls ASP, which processes the requested file from top to bottom, executes any script commands, and sends a Web page to the browser.

Because your scripts run on the server rather than on the client, your Web server does all the work involved in generating the HTML pages sent to browsers. Server-side scripts cannot be readily copied because only the result of the script is returned to the browser. Users cannot view the script commands that created the page they are viewing.

What's New in ASP

Active Server Pages (ASP) has been enhanced with features that make it easier to use for scripters and Web application developers.

- **New Flow Control Capabilities** The ASP **Server** object now has two new methods that you can use to control program flow: **Server.Transfer** and **Server.Execute**. Rather than redirecting requests, which requires an expensive round-trip to the client, you can use these methods to transfer requests directly to an .asp file, without leaving the server.
- **Error Handling** ASP now has a new error-handling capability, so that you can trap errors in a custom error message .asp file. You can use the new **Server.GetLastError** method to display useful information, such as an error description or the line number where the error occurred.
- **Scriptless ASP** Because static content is usually processed more quickly than server-side content, it was previously better to assign an .asp file extension only to files that contained ASP functionality. Whenever you needed to add ASP functionality to your static .html files, you had to manually add .asp file extensions to correct related hyperlinks. With this latest release of ASP, however, .asp files that do not contain server-side functionality are now processed more quickly than ever before. So, if you are creating an evolving Web application in which files may eventually require ASP functionality, you can now conveniently assign those files .asp file extensions, regardless of whether they contain static or server-side content.
- **Performance-Enhanced Objects** ASP now provides performance-enhanced versions of its popular installable components. These objects will scale reliably in a wide range of Web publishing environments.
- **XML Integration** Extensible Markup Language (XML) enables you to semantically describe the complex structure of data or documents, which can then be shared across a variety of applications, clients, and servers. Using the Microsoft® XML Parser, included with Microsoft® Internet Explorer version 4.0, or later, you can create server-side applications enabling your Web server to exchange XML-formatted data with Internet Explorer version 4.0 or later, or with other servers having XML parsing capabilities.
- **Windows Script Components** ASP supports Microsoft's powerful new scripting technology, Windows Script Components. Now you can turn your business logic script procedures into reusable Component Object Model (COM) components that you can use in your Web applications, as well as in other COM-compliant programs.
- **A New Way to Determine Browser Capabilities** ASP has a new feature for determining the exact capabilities of a browser. When a browser sends a cookie describing its capabilities (such a cookie can be installed by using a simple client-side script) you can create an instance of the Browser Capabilities component that retrieves the browser's properties as returned by the cookie. You can use this feature to discover a browser's capabilities and adjust your application accordingly.
- **ASP Self-Tuning** ASP now detects when executing requests are blocked by external resources. It automatically provides more threads to simultaneously execute additional requests and to continue normal processing. If the CPU becomes overburdened, ASP curtails the number of threads. This reduces the constant switching that occurs when too many nonblocking requests are executing simultaneously.
- **Server-Side Include with SRC Attribute** You can now use the HTML <SCRIPT></SCRIPT> tag's SRC attribute to do server-side includes. When you use the SRC attribute to specify a virtual or relative path, and use the RUNAT=SERVER attribute to denote server-side execution, you can achieve the same functionality as the #Include directive.
- **Encoded ASP Scripts** Traditionally, Web developers have been unable to prevent others from viewing the logic behind their scripts. ASP now supports a new script-encoding utility provided with Microsoft® Visual Basic Scripting Edition (VBScript) and Microsoft® JScript 5.0. Web developers can apply an encoding scheme to both client and server-side scripts that makes the programmatic logic appear as unreadable ASCII characters. Encoded scripts are decoded at run time by the script engine, so there's no need for a separate utility. Although this feature is not intended as a secure, encrypted solution, it can prevent most casual users from browsing or copying scripts.
- **International ASP Development** Two new properties were added to the Response object: Response.CodePage and Response.LCID. These properties provide page level control over codepage and locale settings for dynamic strings, without having to enable sessions. Session.CodePage and Session.LCID still provide session level control over codepage and locale settings for dynamic strings; however, they no longer can be inadvertently overwritten by @CodePage and @LCID. @CodePage and @LCID still affect static strings in Web pages. By default, all of these values are implicitly set by the default system ANSI codepage (CP_ACP) and the default system locale (LOCALE_SYSTEM_DEFAULT).

However, if the metabase properties AspCodePage or AspLCID are set for any Web site or virtual directory, these become the default values. The metabase properties allow for global.asa files that need different codepages than the system default. Improved UTF-8 support includes support for surrogate characters and support for localized characters in the intrinsic objects, like form data and server variables.

Important Changes in ASP

ASP has undergone several important changes and enhancements. If you are updating your application from a previous version of ASP, you should be aware of these changes.

- **Buffering On by Default** In IIS 4.0, response buffering was off by default. In IIS 5.1, unless a script specifically turns off buffering, its output is always buffered. This means that the final output is sent to the client at the completion of processing or when the script calls the **Response.Flush** method. When upgrading from IIS 4.0 to IIS 5.1, the IIS 4.0 settings are maintained. Therefore, response buffering remains off until it is turned on.

- **Response.IsClientConnected** In IIS 4.0, **Response.IsClientConnected** returned the correct information only if an ASP file sent content to the browser. In IIS 5.1, an ASP file can use the **IsClientConnected** property prior to sending content to the browser.

- **Improved Include File Security** In IIS 4.0, when an include file resided in a virtual root mapped to a physical path, ASP did not use the security credentials of the physical path to process the file. In IIS 5.1, ASP applies the credentials of the physical path when processing include files.

- **Default Document Query String Behavior** In IIS 5.1, if an .asp (or .cdx) file is configured as the default document, it can now receive a query string from a URL that does not specify the default document. For example, the URLs `http://www.microsoft.com/default.asp?newuser=true` and `http://www.microsoft.com/?newuser=true` will both send a query string value to the default .asp file.

- **Transaction Flags** IIS 4.0 used the **required**, **requires new**, and **not supported** transaction flags to indicate that ASP was starting a new transaction. In IIS 5.1, this behavior is unchanged. However, if an .asp file executes a transacted .asp file using the new **Server.Execute** or **Server.Transfer** methods, then the transaction flag state is maintained for the second .asp file. If the second .asp file's transaction flags indicate that transactions are supported or required, then the existing transaction will be used and a new transaction will not be started.

- **Behavior of Both-Modeled Objects** A Both-Modeled COM object which does not support the Free-Threaded Marshaller will fail if it is stored in Application state. Both-Modeled components must aggregate the Free-Threaded Marshaller to be stored in Application state.

- **Configurable Entries Moved to the Metabase** The following IIS 4.0 registry entries are now in the metabase:
  - ProcessorThreadMax
  - ErrorsToNTLog

  For IIS 5.1, all configurable parameters for ASP can be modified from the metabase.

- **Security for Local Server COM Objects** IIS uses a new Windows COM feature called *cloaking* to enable local server applications instantiated from an .asp file to have the security context of the originating client. In previous versions, the identity assigned to the local server COM object depended on the identity of the user who created the object instance.

- **Objects Released Earlier** In IIS 4.0, COM objects were only released when ASP finished processing a page. In IIS 5.1, if a COM object does not use the **OnEndPage** method and the reference count for the object is zero, then the object is released prior to the completion of processing.

Module 1: Creating ASP Pages

In Module 1, you create ASP pages (.asp files) using HTML and Microsoft® Visual Basic® Scripting Edition (VBScript). This module includes the following lessons:

- Lesson 1: Writing an ASP Page   Four examples of how to write and run ASP pages using HTML and VBScript.
- Lesson 2: Submitting Information Using Forms   Develop a form on an HTML page so that users can enter their personal data.
- Lesson 3: Creating a Guest Book   Use forms to gather information from visitors, store their information in a Microsoft Access database, and display the database contents on a Web page.
- Lesson 4: Displaying an Excel Spreadsheet   Display a Microsoft Excel spreadsheet on a Web page.

Lesson 1: Writing an ASP Page

The best way to learn about ASP is to look at examples; alter integer values, strings, and statements you are curious about; and determine what changes occur in the browser.

In this lesson you perform the following tasks:

- **Example 1**   Create, save, and run an ASP page using HTML and VBScript.
- **Examples 2, 3, and 4**   Add functionality and logic to your ASP page by using built-in functions and conditional script statements.

VBScript is the default scripting language for ASP pages; however, the delimiters are the same for JScript. Use angle brackets as delimiters around HTML tags just as you would in any .htm page, as follows:

```
<example></example>
```

Use percent signs with brackets as delimiters around script code, as follows:

```
<% example %>
```

You can put many script statements inside one pair of script delimiters, as in the following example:

```
<font face="MS Gothic">
<%
 'Create a variable.
 dim strGreeting

 'Set the greeting.
 strGreeting = "Hello World!"

 'Print out the greeting, using the ASP Response object.
 Response.Write strGreeting & "<BR>"

 'Also print out the greeting using the <%= method.
%>
<%=strGreeting%>
</font>
```

This code displays the following text in the browser:

```
Hello World!
Hello World!
```

Here is the previous example using JScript:

```
<%@ Language=JScript %>

<font face="MS Gothic">
<%
 //Create a variable.
 var strGreeting;
```

```
   //Set the greeting.
   strGreeting = "Hello World!";

   //Print out the greeting, using the ASP Response object.
   Response.Write(strGreeting + "<BR>");

   //Also print out the greeting using the <%= method.
  %>
  <%=strGreeting%>
  </font>
```

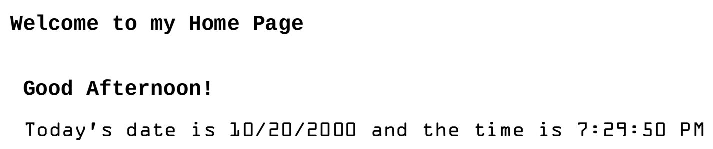
To create ASP pages, use a text editor such as Notepad and save the page with the .asp extension instead of .htm. The .asp filename extension tells IIS to send the page through the ASP engine before sending the page to a client. (**Note:** In the Notepad **Save As** dialog box, when **Text Documents (*.txt)** is selected in the **Save as type** box, Notepad automatically appends .txt to the filename. To prevent this, select **All Files** in the **Save as type** box, type the full filename ***MyFile*.asp** in the **File Name** field, and then click **Save**.)

Example 1

This example displays a greeting, the date, and the current time. To run this example, copy and paste the following code into an empty file and save it in the *x*:\Inetpub\Wwwroot\Tutorial directory as **Example1.asp**. View the example with your browser by typing **http://localhost/Tutorial/Example1.asp** in the address bar.

```
  <%@ Language=VBScript %>

  <html>
  <head>
  <title>Example 1</title>
  </head>
  <body>
  <font face="MS Gothic">

  <H1>Welcome to my Home Page</H1>
  <%
   'Create some variables.
   dim strDynamicDate
   dim strDynamicTime

   'Get the date and time.
   strDynamicDate = Date()
   strDynamicTime = Time()

   'Print out a greeting, depending on the time, by comparing the last 2 characters in
strDymamicTime to "PM".
   If "PM" = Right(strDynamicTime, 2) Then
      Response.Write "<H3>Good Afternoon!</H3>"
   Else
      Response.Write "<H3>Good Morning!</H3>"
   End If
  %>
  Today's date is <%=strDynamicDate%> and the time is <%=strDynamicTime%>

  </font>
  </body>
  </html>
```

In the browser, you should see something like the following (depending on the date and time you perform this exercise):



```
Welcome to my Home Page

Good Afternoon!

Today's date is 10/20/2000 and the time is 7:29:50 PM
```

Note: The Web server processes Example1.asp in the following sequence:

1. **<%@ Language=VBScript %>** tells the ASP engine to use the VBScript engine to translate the script code.
2. The ASP engine ignores the HTML code blocks.
3. The ASP engine executes the code in the **<%...%>** blocks and replaces the blocks with placeholders. The results of the **Response.Write** strings and *<%=...%>* strings are saved in memory on the server.

4. The results of the **Response.Write** strings and *<%=...%>* strings are injected into the HTML code at the matching placeholders before the page leaves the ASP engine.

5. The complete page leaves the ASP engine as a file of HTML code, and the server sends the page to the client.

Example 2

This example incorporates a **For...Next** loop in the ASP page to add a little dynamic logic. The **For...Next** loop is one of six conditional statements available to you. The others are **Do...Loop**, **For Each...Next**, **If...Then...Else...End If**, **Select..Case...End Select**, and **While...Wend**. These statements are documented at Windows Script Technologies under VBScript.

Copy and paste the following code in your text editor, and save the file as **Example2.asp**. View the example with your browser by typing **http://localhost/Tutorial/Example2.asp** in the address bar.

The processing order is the same as in Example1.asp.

```
<%@ Language=VBScript %>

<html>
<head>
<title>Example 2</title>
</head>
<body>
<font face="MS Gothic">

<%
 'Create a variable.
 dim strTemp
 dim font1, font2, font3, fontsize

 'Set the variable.
 strTemp= "BUY MY PRODUCT!"
 fontsize = 0

 'Print out the string 5 times using the For...Next loop.
 For i = 1 to 5

   'Close the script delimiters to allow the use of HTML code and <%=...
   %>.
   <table align=center><font size= <%=fontsize%>> <%=strTemp%> </font></table>
   <%
   fontsize = fontsize + i

 Next

%>
<table align=center><font size=6><B> IT ROCKS! <B></font></table><BR>

</font>
</body>
</html>
```

In the browser, you should see something like the following:

```
BUY MY PRODUCT!

BUY MY PRODUCT!

BUY MY PRODUCT!

BUY MY PRODUCT!

BUY MY PRODUCT!

IT ROCKS!
```

Here is Example 2 using JScript:

```
<%@ Language=JScript %>

<html>
<head>
<title>Example 2</title>
</head>
<body>
<font face="MS Gothic">

<%
//Create a variable.
var strTemp;
var font1, font2, font3, font, size;

//Set the variable.
strTemp= "BUY MY PRODUCT!";
fontsize = 0;

//Print out the string 5 times using the For...Next loop.
for (i = 1; i < 6; i++) {

//Close the script delimiters to allow the use of HTML code and <%=...
%>.
<table align=center><font size= <%=fontsize%>> <%=strTemp%> </font></table>
<%
fontsize = fontsize + i;

}

%>
<table align=center><font size=6><B> IT ROCKS! <B></font></table><BR>

</font>
</body>
</html>
```

Example 3

There are more multilingual Web sites every day, as businesses see the need to offer their products around the world. Formatting your date, time, and currency to match the user's locale is good for diplomacy.

In Example 3, a predefined function displays the date and currency on your ASP page. The date and currency are formatted for different locales using the **GetLocale** function, the **SetLocale** function, the **FormatCurrency** function, and the **FormatDateTime** function. Locale identification strings are listed in the **Locale ID Chart** on MSDN. (This example doesn't cover changing the CodePage to display non-European characters on European operating systems. Please read CodePage topics in your IIS Documentation for more information.)

Note: There are more than 90 predefined functions in VBScript, and they are all well-defined at Windows Script Technologies. To view the documentation, select **VBScript**, select **Documentation**, select **Language Reference**, and select **Functions**.

Copy and paste the following code in your text editor, and save the file as **Example3.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/Example3.asp** in the address bar.

```
<%@ Language=VBScript %>

<html>
<head>
<title>Example 3</title>
</head>
<body>
<font face="MS Gothic">

<H3>Thank you for your purchase.  Please print this page for your records.</H3>
<%
 'Create some variable.
 dim saveLocale
 dim totalBill

 'Set the variables.
 saveLocale = GetLocale
 totalBill = CCur(85.50)

 'For each of the Locales, format the date and the currency
 SetLocale("fr")
 Response.Write"<B>Formatted for French:</B><BR>"
 Response.Write FormatDateTime(Date, 1) & "<BR>"
```

```
   Response.Write FormatCurrency(totalBill) & "<BR>"
   SetLocale("de")
   Response.Write"<B>Formatted for German:</B><BR>"
   Response.Write FormatDateTime(Date, 1) & "<BR>"
   Response.Write FormatCurrency(totalBill) & "<BR>"
   SetLocale("en-au")
   Response.Write"<B>Formatted for English - Australia:</B><BR>"
   Response.Write FormatDateTime(Date, 1)& "<BR>"
   Response.Write FormatCurrency(totalBill) & "<BR>"

   'Restore the original Locale
   SetLocale(saveLocale)
  %>

  </font>
  </body>
  </html>
```

In the browser, you should see the following:

```
Thank you for your purchase. Please print this page for your records.

  Formatted for French:
  vendredi 20 octobre 2000
  85,50 F
  Formatted for German:
  Freitag, 20. Oktober 2000
  85,50 DM
  Formatted for English - Australia:
  Friday, 20 October 2000
  $85.50
```

Example 4

The most common functions used in ASP Scripts are those that manipulate strings. The most powerful string functions use regular expressions. Because regular expressions are difficult to adapt to, Example 4 shows you how to replace characters in a string by using a string expression and a regular expression. Regular expressions are defined at Windows Script Technologies. To view the documentation, select **VBScript**, select **Documentation**, and select **Regular Expressions Guide**.

Copy and paste the following code in your text editor, and save the file as **Example4.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/Example4.asp** in the address bar.

```
  <%@ Language=VBScript %>

  <html>
  <head>
  <title>Example 4</title>
  </head>
  <body>
  <font face="MS Gothic">

  <H3>Changing a Customer's Street Address</H3>
  <%
   'Create some variables.
   dim strString
   dim strSearchFor     ' as a string
   dim reSearchFor     ' as a regular expression
   dim strReplaceWith

   'Set the variables.
   strString = "Jane Doe<BR>100 Orange Road<BR>Orangeville,
WA<BR>98100<BR>800.555.1212<BR>"
   '   Using a string object
   strSearchFor = "100 Orange Road<BR>Orangeville, WA<BR>98100"
   '   Using a regular expression object
   Set reSearchFor = New RegExp
   reSearchFor.Pattern = "100 Orange Road<BR>Orangeville, WA<BR>98100"
   reSearchFor.IgnoreCase = False

   strReplaceWith = "200 Bluebell Court<BR>Blueville, WA<BR>98200"

   'Verify that strSearchFor exists...
   '   using a string object.
   If Instr(strString, strSearchFor) Then
     Response.Write "strSearchFor was found in strString<BR>"
   Else
     Response.Write "Fail"
   End If
   '   using a regular expression object.
   If reSearchFor.Test(strString) Then
```

```
    Response.Write "reSearchFor.Pattern was found in strString<BR>"
  Else
    Response.Write "Fail"
  End If

  'Replace the string...
  Response.Write "<BR>Original String:<BR>" & strString & "<BR>"
  '   using a string object.
  Response.Write "String where strSearchFor is replaced:<BR>"
  Response.Write Replace(strString, strSearchFor, strReplaceWith) & "<BR>"
  '   using a regular expression object.
  Response.Write "String where reSearchFor is replaced:<BR>"
  Response.Write reSearchFor.Replace(strString, strReplaceWith) & "<BR>"
 %>

 </font>
 </body>
 </html>
```

In the browser, you should see the following:

```
Changing a Customer's Street Address

 strSearchFor was found in strString
 reSearchFor.Pattern was found in strString

 Original String:
 Jane Doe
 100 Orange Road
 Orangeville, WA
 98100
 800.555.1212

 String where strSearchFor is replaced:
 Jane Doe
 200 Bluebell Court
 Blueville, WA
 98200
 800.555.1212

 String where reSearchFor is replaced:
 Jane Doe
 200 Bluebell Court
 Blueville, WA
 98200
 800.555.1212
```

Lesson 2: Submitting Information Using Forms

A common use of intranet and Internet server applications is to accept user input by implementing a form in your Web page. ASP includes the following two collections in the **Request** object to help process form information: the **QueryString** collection and the **Form** collection.

In this lesson, you create an HTML page that accepts user input in an HTML form and sends the user input back to the Web server to the same page. The Web server then displays the user input. Later in this module, you use this knowledge about forms to build a guest book application that uses ASP scripting. To complete this lesson, you perform the following tasks:

- **Example 1** Display a selection of button elements in a form.
- **Example 2** Display text box elements in a form, accept the user input from the form, and display the user input on the Web page.

Example 1: Buttons

Forms can contain many different kinds of elements to help your users enter data. In this example, there are five input form elements called buttons. There are many types of buttons including **RADIO** buttons, **SUBMIT** buttons, **RESET** buttons, **CHECKBOX** buttons, and **TEXT** buttons.

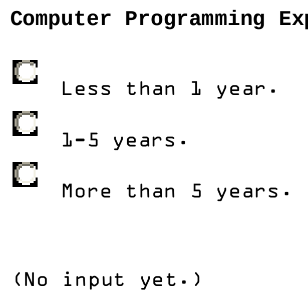
After the user enters information in a form, the information needs to be sent to your Web application. When a user clicks the button labeled "Submit" in your Web page, the form data is sent from the client to the Web page that is listed in the **ACTION** element of the form tag. This Web page doesn't need to be the same as the calling page. In this example, the Web page listed in the **ACTION** element is the same as the calling page, which eliminates the need to call another page.

In this example, **METHOD**="**POST**" is used to send data from the Web client's browser to the Web server. When you use **METHOD**="**POST**" in a form, the user data ends up in the **Form** collection of the **Request** object.

Copy and paste the following code in your text editor, and save the file as **Button.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/Button.asp** in the address bar.

```
<%@ Language=VBScript %>

<html>
<head>
<title>Button Form</title>
</head>
<body>
<font face="MS Gothic">

<FORM NAME="Button Example" METHOD="POST" ACTION="button.asp">
<H3>Computer Programming Experience:</H3>
<p>
<INPUT TYPE="RADIO" NAME= "choice" VALUE="Less than 1"> Less than 1 year.<BR>
<INPUT TYPE="RADIO" NAME= "choice" VALUE="1 to 5"> 1-5 years.<BR>
<INPUT TYPE="RADIO" NAME= "choice" VALUE="More than 5"> More than 5 years.<BR>
</p>
<p>
<INPUT TYPE="SUBMIT" VALUE="Submit">
<INPUT TYPE="RESET" VALUE="Clear Form">
</p>
</form>

<%
 'Check to see if input has already been entered.
 dim strChoice
 strChoice = Request.Form("choice")

 If "" = strChoice Then
   Response.Write "<P>(No input yet.)</P>"
 Else
   Response.Write "<P>Your last choice was <B>" & strChoice & "</B></P>"
 End If
%>

</font>
</body>
</html>
```

In the browser, you should see the following:



Computer Programming Experience:

Less than 1 year.

1-5 years.

More than 5 years.

(No input yet.)

Example 2: Input Form Elements

In this example, there are three input form elements called text fields and two input form elements called check boxes. Check boxes differ from option buttons because you can select more than one. We still need the default **Submit** button to send the data back to the server.

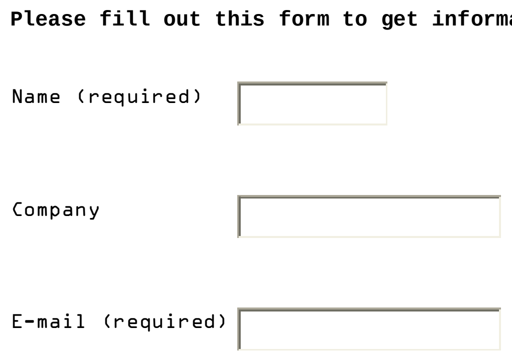
In this example, METHOD=GET is used to send data from the Web client's browser to the Web server. When you use METHOD=GET in a form, the user data ends up in the **QueryString** collection of the **Request** object.

Look at the address bar after you click **Submit**, and you should see the elements of the **QueryString** displayed at the end of the URL.

Copy and paste the following code in your text editor, and save the file as **Text.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/Text.asp** in the address bar.

```
<%@ Language=VBScript %>

<html>
<head>
<title>Text and Checkbox Form</title>
</head>
<body>
<font face="MS Gothic">

<FORM NAME="TextCheckbox Example" METHOD="GET" ACTION="text.asp">
<H3>Please fill out this form to get information on our products:</H3>
<p>
<table>
<tr>
<td><font face="MS Gothic">Name (required)</td>
<td><INPUT TYPE="TEXT" NAME="name" VALUE="" SIZE="20" MAXLENGTH="150"></td>
</tr><tr>
<td><font face="MS Gothic">Company</td>
<td><INPUT TYPE="TEXT" NAME="company" VALUE="" SIZE="25" MAXLENGTH="150"></td>
</tr><tr>
<td><font face="MS Gothic">E-mail (required)</td>
<td><INPUT TYPE="TEXT" NAME="email" VALUE="" SIZE="25" MAXLENGTH="150"></td>
</tr>
</table>
</p>
<p>
Requesting information on our:<BR>
<INPUT TYPE="CHECKBOX" NAME= "info" VALUE="software">Software<BR>
<INPUT TYPE="CHECKBOX" NAME= "info" VALUE="hardware">Hardware <BR>
</p>
<p>
<INPUT TYPE="SUBMIT" VALUE="Submit">
<INPUT TYPE="RESET" VALUE="Clear Form">
</p>
</form>

<%
 'Check to see if input has already been entered.
 dim strName, strEmail, strCompany, strInfo
 strName = Request.QueryString("name")
 strEmail = Request.QueryString("email")
 strCompany = Request.QueryString("company")
 strInfo = Request.QueryString("info")

 'Display what was entered.
 If ("" = strName) OR ("" = strEmail) Then
   Response.Write "<P>(No required input entered yet.)</P>"
 Else
   Response.Write "<P>Your are " & strName
   If Not ("" = strCompany) Then
     Response.Write " from " & strCompany
   End If
   Response.Write ". <BR>Your email address is " & strEmail
   If Not ("" = strInfo) Then
     Response.Write " and you would like information on " & strInfo & ".</P>"
   End If
 End If
%>

</font>
</body>
</html>
```

In the browser, you should see the following:



Please fill out this form to get information about our products:

Name (required)	
Company	
E-mail (required)	

```
Requesting information about our:
☐
    Software
☐
    Hardware


(No required input entered yet.)
```

Lesson 3: Creating a Guest Book using a Database

This lesson requires that you have Microsoft Access installed on your system and is not supported on 64-bit platforms until Access is developed for 64-bit platforms.

Lesson 3 develops a guest book application. Guest books allow your Web site visitors to leave behind information, such as their names, e-mail addresses, and comments. In this lesson, you perform the following tasks after creating an Access database:

- **Example 1** Create an ASP page to connect to the database using only the ADO **Connection** object.
- **Example 2** Create an ASP page to connect to the database using the **Connection** object and the **Command** object together.
- **Example 3** Create an ASP page to display the guest book information from the database in a browser.

Create an Access Database

Create an Access database called **GuestBook.mdb**, and save it in *x*:\Inetpub\Wwwroot\Tutorial. Create a table in the database called **GuestBook**. Use the **Create Table Using Design View** option in Access to add the following fields and properties:

Field Name	**Data Type**	**Field General Properties**
FID	AutoNumber	Field Size=Long Integer, New Values=Increment, Indexed=Yes(No Duplicates)
FTB1	Text	Field Size=255, Required=No, Allow Zero Length=Yes, Indexed=No
FTB2	Text	Field Size=255, Required=No, Allow Zero Length=Yes, Indexed=No
FTB3	Text	Field Size=255, Required=No, Allow Zero Length=Yes, Indexed=No
FTB4	Text	Field Size=255, Required=No,

		Allow Zero Length=Yes, Indexed=No
FMB1	Memo	Required=No, Allow Zero Length=Yes

Create an ASP Page to Add Data to Your Access Database

Now that you have created a database, you can build an ASP page to connect to your database and read incoming data using Microsoft ActiveX® Data Objects (ADO). ADO is a collection of objects with methods and properties that can manipulate data in almost any type of database. (If you plan to use databases frequently, you should purchase a programmer's reference book for ADO. Only the most basic ADO code is illustrated in the following examples, enough to open, read in, and write to a database.)

The next two examples produce the same results; however, the first example uses only the **Connection** object, and the second example gives part of the job to the **Command** object, which is much more powerful. Compare both examples to see how the objects become connected together. After you are comfortable with the the objects, use an ADO programmer's reference to experiment with more methods and properties.

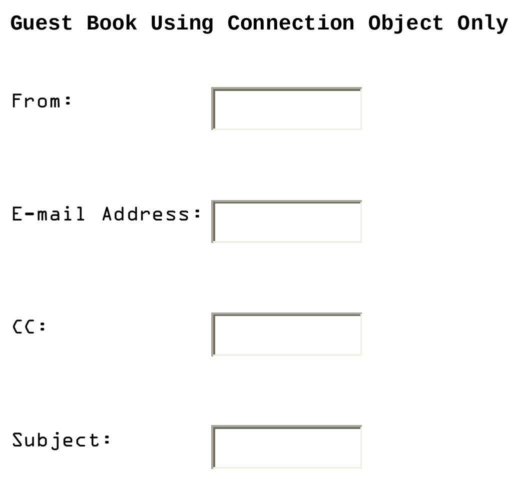
To see an example of an ADO error in your ASP page, try browsing to the page after renaming your database, after entering a typing mistake in the connection string, or after making the database Read Only.

Example 1: Using Only the ADO Connection Object

Copy and paste the following code in your text editor, and save the file as **GuestBook1.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/GuestBook1.asp** in the address bar.

```
  <%@ Language=VBScript %>

  <html>
  <head>
  <title>Guest Book Using Connection Object Only</title>
  </head>
  <body>
  <font face="MS Gothic">
  <h2>Guest Book Using Connection Object Only</h2>

  <%
   If Not Request.QueryString("Message") = "True" Then
     'No information has been input yet, so provide the form.
  %>
     <p>
     <FORM NAME="GuestBook1" METHOD="GET" ACTION="guestbook1.asp">
     <table>
       <tr>
       <td><font face="MS Gothic">From:</td><td><INPUT TYPE="TEXT" NAME="From"></td>
       </tr><tr>
       <td><font face="MS Gothic">E-mail Address:</td><td><INPUT TYPE="TEXT"
NAME="EmailAdd"></td>
       </tr><tr>
       <td><font face="MS Gothic">CC:</td><td><INPUT TYPE="TEXT" NAME="CC"></td>
       </tr><tr>
       <td><font face="MS Gothic">Subject:</td><td><INPUT TYPE="TEXT"
NAME="Subject"></td>
       </tr>
     </table>
     Message:<br><TEXTAREA NAME="Memo" ROWS=6 COLS=70></TEXTAREA>
     </p>

     <p>
     <INPUT TYPE="HIDDEN" NAME="Message" VALUE="True">
     <INPUT TYPE="SUBMIT" VALUE="Submit Information">
     </FORM>
     </p>
  <%
   Else
     'The HIDDEN button above sets the Message variable to True.
     'We know now that form data has been entered.

     'Get the data from the form. We will be inserting it into the database.
     'Access doesn't like some characters, such as single-quotes, so encode the
     ' data using the HTMLEncode method of the ASP Server object.
     dim strTB1, strTB2, strTB3, strTB4, strMB1, strCommand
     strTB1 = Server.HTMLEncode(Request.QueryString("From"))
     strTB2 = Server.HTMLEncode(Request.QueryString("EMailAdd"))
     strTB3 = Server.HTMLEncode(Request.QueryString("CC"))
     strTB4 = Server.HTMLEncode(Request.QueryString("Subject"))
     strMB1 = Server.HTMLEncode(Request.QueryString("Memo"))

     'This is a connection string. ADO uses it to connect to a database through the
Access driver.
     'It needs the provider name of the Access driver and the name of the Access
database.
     'Connection strings are slightly different, depending on the provider being used,
     ' but they all use semicolons to separate variables.
     'If this line causes and error, search in your registry for
     ' Microsoft.JET to see if 4.0 is your version.
     strProvider = "Provider=Microsoft.JET.OLEDB.4.0;Data
Source=C:\InetPub\Wwwroot\Tutorial\guestbook.mdb;"

     'This creates an instance of an ADO Connection object.
     'There are 4 other ADO objects available to you, each with different methods and
     'properties that allow you to do almost anything with database data.
     Set objConn = server.createobject("ADODB.Connection")

     'The Open method of the Connection object uses the connection string to
     ' create a connection to the database.
     objConn.Open strProvider

     'Define the query.
     'There are many types of queries, allowing you to add, remove, or get data.
     'This query will add your data into the database, using the INSERT INTO key words.
     'Here, GuestBook is the name of the table.
     'You need single-quotes around strings here.
     strCommand = "INSERT INTO GuestBook (FTB1,FTB2,FTB3,FTB4,FMB1) VALUES ('"
     strCommand = strCommand & strTB1 & "','" & strTB2 & "','" & strTB3 & "','" & strTB4
& "','" & strMB1
     strCommand = strCommand & "')"

     'Execute the query to add the data to the database.
     objConn.Execute strCommand

     Response.Write("Thank you! Your data has been added.")

     End If
  %>

  </font>
  </body>
  </html>
```

In the browser, you should see the following:



Guest Book Using Connection Object Only

From:

E-mail Address:

CC:

Subject:

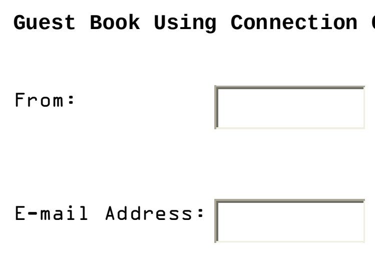
Example 2: Using the Connection Object and the Command Object Together

Copy and paste the following code in your text editor, and save the file as **GuestBook2.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/GuestBook2.asp** in the address bar.

```
  <%@ Language=VBScript %>

  <html>
  <head>
  <title>Guest Book Using Connection Object and Command Object</title>
  </head>
  <body>
  <font face="MS Gothic">
  <h2>Guest Book Using Connection Object and Command Object</h2>

  <%
   If Not Request.QueryString("Message") = "True" Then
     'No information has been input yet, so provide the form.
  %>
     <p>
     <FORM NAME="GuestBook2" METHOD="GET" ACTION="guestbook2.asp">
     <table>
       <tr>
       <td><font face="MS Gothic">From:</td><td><INPUT TYPE="TEXT" NAME="From"></td>
       </tr><tr>
       <td><font face="MS Gothic">E-mail Address:</td><td><INPUT TYPE="TEXT"
NAME="EmailAdd"></td>
       </tr><tr>
       <td><font face="MS Gothic">CC:</td><td><INPUT TYPE="TEXT" NAME="CC"></td>
       </tr><tr>
       <td><font face="MS Gothic">Subject:</td><td><INPUT TYPE="TEXT"
NAME="Subject"></td>
       </tr>
     </table>
     Message:<br><TEXTAREA NAME="Memo" ROWS=6 COLS=70></TEXTAREA>
     </p>

     <p>
     <INPUT TYPE="HIDDEN" NAME="Message" VALUE="True">
     <INPUT TYPE="SUBMIT" VALUE="Submit Information">
     </FORM>
     </p>
  <%
   Else
     'The HIDDEN button above sets the Message variable to True.
     'We know now that form data has been entered.

     'Get the data from the form. We will be inserting it into the database.
     'Access doesn't like some characters, such as single-quotes, so encode the
     ' data using the HTMLEncode method of the ASP Server object.
     dim strTB1, strTB2, strTB3, strTB4, strMB1
     strTB1 = Server.HTMLEncode(Request.QueryString("From"))
     strTB2 = Server.HTMLEncode(Request.QueryString("EMailAdd"))
     strTB3 = Server.HTMLEncode(Request.QueryString("CC"))
     strTB4 = Server.HTMLEncode(Request.QueryString("Subject"))
     strMB1 = Server.HTMLEncode(Request.QueryString("Memo"))

     'The Memo data type in the Access database allows you to set the field size.
     If strMB1 = "" Then
       iLenMB1 = 255
     Else
       iLenMB1 = Len(strMB1)
     End If

     'This is a connection string. ADO uses it to connect to a database through the
Access driver.
     'It needs the provider name of the Access driver and the name of the Access
database.
```

```
     'Connection strings are slightly different, depending on the provider being used,
     ' but they all use semicolons to separate variables.
     'If this line causes and error, search in your registry for
     ' Microsoft.JET to see if 4.0 is your version.
     strProvider = "Provider=Microsoft.JET.OLEDB.4.0;Data
Source=C:\InetPub\Wwwroot\Tutorial\guestbook.mdb;"

     'This creates an instance of an ADO Connection object.
     'There are 4 other ADO objects available to you, each with different methods and
     'properties that allow you to do almost anything with database data.
     Set objConn = server.createobject("ADODB.Connection")

     'The Open method of the Connection object uses the connection string to
     ' create a connection to the database.
     objConn.Open strProvider

     'This creates an instance of an ADO Command object.
     'Although you could do most of your work with the Connection object,
     ' the Command object gives you more control.
     Set cm = Server.CreateObject("ADODB.Command")

     'The ActiveConnection property allows you to attach to an Open connection.
     'This is how you link the Connection object above to the Command object.
     cm.ActiveConnection = objConn

     'Define the query.
     'There are many types of queries, allowing you to add, remove, or get data.
     'This query will add your data into the database, using the INSERT INTO keywords.
     'Because we are using the Command object, we need to put our query into the
     ' CommandText property.
     'Here, GuestBook is the name of the table.
     cm.CommandText = "INSERT INTO GuestBook (FTB1,FTB2,FTB3,FTB4,FMB1) VALUES
(?,?,?,?,?)"

     'This is where you see the power of the Command object.
     'By putting ? marks in the string above, we can use the Parameters collection
     ' to have ADO fill in the ? with the detailed parameters below.
     'cm.CreateParameter formats the parameter for you.
     'cm.Parameters.Append appends the parameter to the collection.
     'Make sure they are in the same order as (TB1,TB2,TB3,TB4,MB1).

     Set objparam = cm.CreateParameter(, 200, , 255, strTB1)
     cm.Parameters.Append objparam

     Set objparam = cm.CreateParameter(, 200, , 255, strTB2)
     cm.Parameters.Append objparam

     Set objparam = cm.CreateParameter(, 200, , 255, strTB3)
     cm.Parameters.Append objparam

     Set objparam = cm.CreateParameter(, 200, , 255, strTB4)
     cm.Parameters.Append objparam

     Set objparam = cm.CreateParameter(, 201, , iLenMB1, strMB1)
     cm.Parameters.Append objparam

     'Execute the query to add the data to the database.
     'Here, the Execute method of the Command object is called instead of
     ' the Execute method of the Connection object.
     cm.Execute

     Response.Write("Thank you! Your data has been added.")

     End If
  %>

  </font>
  </body>
  </html>
```

In the browser, you should see the same content as GuestBook1.asp, as follows:



Guest Book Using Connection Object Only

From:

E-mail Address:

Example 3: Display the Database in a Browser

After information is entered in a database, you can use a Web page containing another script to view and edit the data. Not much changes in the ADO code, except the way you define your query.
In the last two examples, you used the **INSERT INTO** query to add records to a database. In this example, you use the **SELECT** query to choose records from a database and display them in the browser. You also use the **DELETE** query to remove records from the database. The only query not used in these examples is the **UPDATE** query, whose syntax is like that of the **INSERT INTO** query. The **UPDATE** query allows you to change fields in a database.

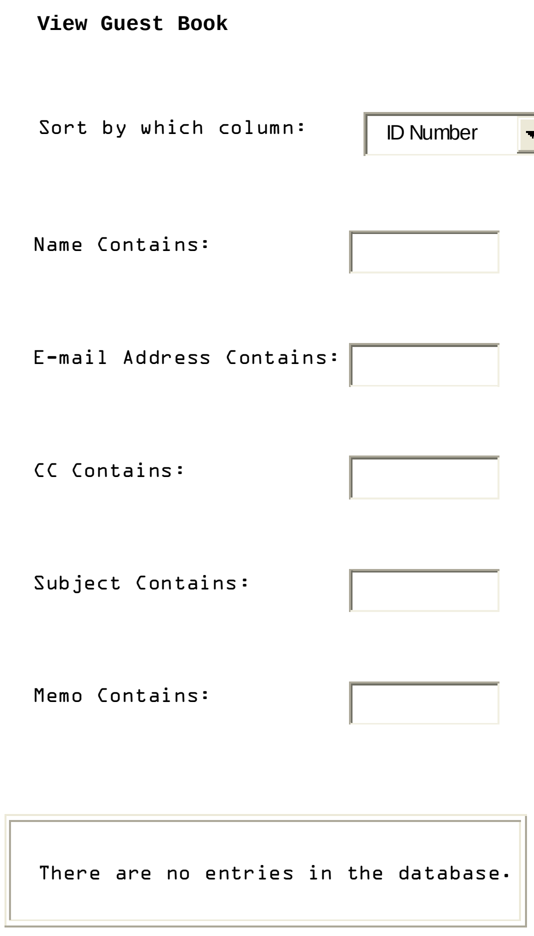
Copy and paste the following code in your text editor, and save the file as **ViewGB.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing
http://localhost/Tutorial/ViewGB.asp
in the address bar.

```
<%@ Language=VBScript %>

<html>
<head>
<title>View Guest Book</title>
</head>
<body>
<font face="MS Gothic">
<h2>View Guest Book</h2>

<%
'Read in any user input. Any of these can be empty.
'By doing this first, we can preserve the user input in the form.
dim strTB1, strTB2, strTB3, strTB4, strMB1, strSort, iDelete
strTB1 = Server.HTMLEncode(Request.QueryString("From"))
strTB2 = Server.HTMLEncode(Request.QueryString("EMailAdd"))
strTB3 = Server.HTMLEncode(Request.QueryString("CC"))
strTB4 = Server.HTMLEncode(Request.QueryString("Subject"))
strMB1 = Server.HTMLEncode(Request.QueryString("Memo"))
strSort = Server.HTMLEncode(Request.QueryString("sort"))
iDelete = CInt(Request.QueryString("Delete"))

'Because we use this variable, and it might not be set...
If "" = strSort Then
  strSort = "FID"
End If
%>

<p>
<FORM NAME="ViewGuestBook" METHOD="GET" ACTION="viewgb.asp">
<table>
  <tr>
  <td><font face="MS Gothic">Sort by which column:</td>
  <td><SELECT NAME="sort" SIZE="1">
    <OPTION VALUE="FID">ID Number</OPTION>
    <OPTION VALUE="FTB1">Name</OPTION>
    <OPTION VALUE="FTB2">Email</OPTION>
    <OPTION VALUE="FTB3">CC</OPTION>
    <OPTION VALUE="FTB4">Subject</OPTION>
    <OPTION VALUE="FMB1">Memo</OPTION>
  </SELECT></td>
  </tr><tr>
```

```
    <td><font face="MS Gothic">Name Contains:</td>
    <td><INPUT TYPE="TEXT" NAME="From" VALUE="<%=strTB1%>"></td>
    </tr><tr>
    <td><font face="MS Gothic">E-mail Address Contains:</td>
    <td><INPUT TYPE="TEXT" NAME="EmailAdd" VALUE="<%=strTB2%>"></td>
    </tr><tr>
    <td><font face="MS Gothic">CC Contains:</td>
    <td><INPUT TYPE="TEXT" NAME="CC" VALUE="<%=strTB3%>"></td>
    </tr><tr>
    <td><font face="MS Gothic">Subject Contains:</td>
    <td><INPUT TYPE="TEXT" NAME="Subject" VALUE="<%=strTB4%>"></td>
    </tr><tr>
    <td><font face="MS Gothic">Memo Contains:</td>
    <td><INPUT TYPE="TEXT" NAME="Memo" VALUE="<%=strMB1%>"></td>
    </tr>
  </table>
  <INPUT TYPE="SUBMIT" VALUE="Submit Search Parameters">
  </p>

  <%
  'Create your connection string, create an instance of the Connection object,
  ' and connect to the database.
  strProvider = "Provider=Microsoft.JET.OLEDB.4.0;Data
Source=C:\InetPub\Wwwroot\Tutorial\guestbook.mdb;"
  Set objConn = Server.CreateObject("ADODB.Connection")
  objConn.Open strProvider

  'Define the query.
  If iDelete = 0 Then
    'If the Delete variable is not set, the query is a SELECT query.
    '* means all fields. ASC means ASCII. % is a wildcard character.
    strQuery = "SELECT * FROM GuestBook"
    strQuery = strQuery & " WHERE FTB1 LIKE '%" & strTB1 & "%'"
    strQuery = strQuery & " AND FTB2 LIKE '%" & strTB2 & "%'"
    strQuery = strQuery & " AND FTB3 LIKE '%" & strTB3 & "%'"
    strQuery = strQuery & " AND FTB4 LIKE '%" & strTB4 & "%'"
    strQuery = strQuery & " AND FMB1 LIKE '%" & strMB1 & "%'"
    strQuery = strQuery & " ORDER BY " & StrSort & " ASC"
  Else
    'We want to delete a record.
    strQuery = "DELETE FROM GuestBook WHERE FID=" & iDelete
  End If

  'Executing the SELECT query creates an ADO Recordset object.
  'This holds the data you get from the database.
  Set objRS = objConn.Execute(strQuery)

  'Now that you have the database data stored in the Recordset object,
  ' show it in a table.
 %>

  <p>
  <FORM NAME="EditGuestBook" METHOD="GET" ACTION="viewgb.asp">
  <table border=1 cellpadding=4 >
  <%
    On Error Resume Next

    If objRS.EOF Then
      If iDelete = 0 Then
        Response.Write "<tr><td><font face=&quot;MS Gothic&quot;>There are no entries in
the database.</font></td></tr>"
      Else
        Response.Write "<tr><td><font face=&quot;MS Gothic&quot;>Record " & iDelete & "
was deleted.</font></td></tr>"
      End If
    Else

      'Print out the field names using some of the methods and properties
      ' of the Recordset object.
      Response.Write "<tr>"

      'For each column in the current row...
      For i = 1 to (objRS.Fields.Count - 1)
          ' write out the field name.
          Response.Write "<td><font face=&quot;MS Gothic&quot;><B>" & objRS(i).Name &
"</B></font></td>"
      Next

      Response.Write "<td><font face=&quot;MS Gothic&quot;><B>Delete</B></font></td>"
      Response.Write "</tr>"

      'Print out the field data, using some other methods and properties
      ' of the Recordset object. When you see a pattern in how they are used,
      ' you can look up others and experiment.

      'While not at the end of the records in the set...
      While Not objRS.EOF
```

```
        Response.Write "<tr>"

        'For each column in the current row...
        For i = 1 to (objRS.Fields.Count - 1)
          ' write out the data in the field.
          Response.Write "<td><font face=&quot;MS Gothic&quot;>" & objRS(i) &
"</font></td>"
        Next

        'Add a button that will pass in an ID number to delete a record.
        %><td><INPUT TYPE="SUBMIT" NAME="Delete" VALUE="<%=objRS(0)%>"></td><%

        Response.Write "</tr>"

        'Move to the next row.
        objRS.MoveNext

      Wend

    End If   'objRS.EOF
  %>
  </table>
  </FORM>

  <%
  'Close the Connection.
  objConn.Close
  %>

  </font>
  </body>
  </html>
```

In the browser, you should see the following:



View Guest Book

Sort by which column:	ID Number
Name Contains:	
E-mail Address Contains:	
CC Contains:	
Subject Contains:	
Memo Contains:	

There are no entries in the database.

Lesson 4: Displaying an Excel Spreadsheet

This lesson requires that you have Microsoft Excel installed on your system and is not supported on 64-bit platforms until Excel is developed for 64-bit platforms.

This lesson demonstrates how to display a Microsoft Excel spreadsheet in a Web page. As in the previous lesson, you use ADO. However, in this lesson you connect to an Excel spreadsheet instead of an Access database.

Prepare your Excel spreadsheet to display in an Active Server Page

1. Using either Excel 98 or Excel 2000, create a spreadsheet and save it as **ASPTOC.xls** in *x*:\Inetpub\Wwwroot\Tutorial. Do not include any special formatting or column labels when creating the spreadsheet.
2. Fill in some of the fields with random data. Treat the first row of cells as column names.
3. Highlight the rows and columns on the spreadsheet that you want to display in the Web page. (Click on one of the fields and drag your mouse diagonally to highlight a block of fields.)
4. On the **Insert** menu, select **Name**, and click **Define**. This is where all your workbooks are defined by name and range of cells.
5. Make sure the cell range you highlighted is displayed correctly at the bottom. Type the name **MYBOOK** for your workbook, and select **Add**. Whenever you change MYBOOK, make sure the correct cell range is displayed in the **Refers to** text box at the bottom of the **Define Name** window. Simply selecting MYBOOK after highlighting a new block of cells does not update the cell range.
6. If the name shows up in the workbook list, click **OK**. Save your spreadsheet.
7. Close Excel to remove the lock on the file so that your ASP page can access it.

In the examples in Lesson 3, you specified a provider name in the connection string, which maps to a specific ADO DLL. In this example, you use a driver name, which causes ASP to use the default provider for that driver name.

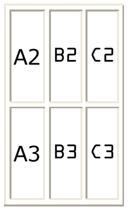
Copy and paste the following code in your text editor, and save the file as **ViewExcel.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/ViewExcel.asp** in the address bar.

```
  <%@ Language=VBScript %>

  <html>
  <head>
  <title>Display an Excel Spreadsheet in a Web Page</title>
  </head>
  <body>
  <font face="MS Gothic">
  <h2>Display an Excel Spreadsheet in a Web Page</h2>

  <%
   'Create your connection string, create an instance of the Connection object,
   ' and connect to the database.
   strDriver = "Driver={Microsoft Excel Driver
(*.xls)};DBQ=C:\Inetpub\Wwwroot\Tutorial\MyExcel.xls;"
   Set objConn = Server.CreateObject("ADODB.Connection")
   objConn.Open strDriver

   'Selects the records from the Excel spreadsheet using the workbook name you saved.
   strSELECT = "SELECT * from `MYBOOK`"

   'Create an instance of the ADO Recordset object, and connect it to objConn.
   Set objRS = Server.CreateObject("ADODB.Recordset")
   objRS.Open strSELECT, objConn

   'Print the cells and rows in the table using the GetString method.
   Response.Write "<H4>Get Excel data in one string with the GetString method</H4>"
   Response.Write "<table border=1 ><tr><td>"
   Response.Write objRS.GetString (, , "</td><td><font face=&quot;MS Gothic&quot;>",
"</font></td></tr><tr><td>", NBSPACE)
   Response.Write "</td></tr></table>"

   'Move to the first record.
   objRS.MoveFirst

   'Print the cells and rows in the table using the ViewGB.asp method.
   Response.Write "<H4>Get Excel data using MoveNext</H4>"

   'Print out the field names using some of the methods and properties
   ' of the Recordset object.
   Response.Write "<table border=1 ><tr>"
   'For each column in the current row...
   For i = 0 to (objRS.Fields.Count - 1)
     ' write out the field name.
     Response.Write "<td><font face=&quot;MS Gothic&quot;><B>" & objRS(i).Name &
"</B></font></td>"
   Next

   'While not at the end of the records in the set...
   While Not objRS.EOF

     Response.Write "</tr><tr>"

     'For each column in the current row...
     For i = 0 to (objRS.Fields.Count - 1)
       ' write out the data in the field.
       %><td><font face="MS Gothic"><%=objRS(i)%></font></td><%
     Next

     'Move to the next row.
     objRS.MoveNext

   Wend

   Response.Write "</tr></table>"

   'Close the Connection.
   objConn.Close
  %>

  </font>
  </body>
  </html>
```

In the browser, you should see the following:

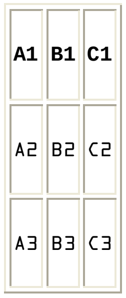
Display an Excel Spreadsheet in a Web Page

Get Excel data in one string with the GetString method



A2	B2	C2
A3	B3	C3

Get Excel data using MoveNext



A1	B1	C1
A2	B2	C2
A3	B3	C3

Up Next: Using COM Components in ASP Pages

Module 2: Using COM Components in ASP Pages

Using Microsoft® Component Object Model (COM) components in your ASP pages can dramatically extend the power that ASP already offers. COM components are pieces of compiled code that can be called from ASP pages. COM components are secure, compact, reusable objects that are compiled as DLLs and can be written in Microsoft Visual C++®, Microsoft Visual Basic®, or any language that supports COM. For example, Microsoft ActiveX® Data Objects (ADO), which are used in Module 1 of this tutorial, provides you with methods and properties to efficiently query databases. By using ADO, you don't have to write your own complex data access code because ADO objects are built using COM components.

In this module, you use COM components that are included with ASP, and you also have a chance to write one of your own. This module shows how to develop an ASP page that delivers services useful in e-commerce and includes the following lessons:

- Lesson 1: Rotating Advertisements   Rotate ads on a Web page, record user data, and redirect users when advertisements are clicked on.
- Lesson 2: Counting Page Hits   Track the number of times users request a page.
- Lesson 3: Creating a Visual Basic COM Object   Create your own ActiveX object using Microsoft Visual Basic.
- Lesson 4: Creating a Java COM Object   Create your own Java object using Microsoft Visual J++®.

Lesson 1: Rotating Advertisements

Advertising is big business on the Web. This lesson describes how to use the Ad Rotator component, installed with IIS, to rotate advertisements on your Web pages. The Ad Rotator component selects an advertisement for your Web page each time the user loads or refreshes the Web page. Furthermore, if you want to change an advertisement, you only need to change it in the redirection and rotation schedule files, instead of changing all the ASP files that contain the advertisement. This saves development time if the advertisement appears on numerous pages within your Web site.

Two files are required to set up the Ad Rotator component: a redirection file, which contains URL links to ads, and a rotation schedule file, which contains display data. By setting up these two files, the Ad Rotator component can be called by any ASP page on your Web site.

In this lesson you perform the following tasks:

- **Example 1**   Create an Ad Rotator rotation schedule file that creates ad-image links on any page that calls this file.
- **Example 2**   Create an Ad Rotator redirection file that specifies global ad-display data and information specific to each advertisement.
- **Example 3**   Create an include file to hold your Ad Rotator calling code.
- **Example 4**   Test the Ad Rotator by creating an ASP page that calls the Ad Rotator component to display and rotate ads. This example requires that examples 1, 2, and 3 are completed first.

Example 1: Create an Ad Rotator Rotation Schedule File

A rotation schedule file is used to catalog information about the ads you want to display, including redirection information after the advertisement is clicked on, the size of the displayed advertisement, the image to display, a comment for the advertisement, and a number that indicates how often a particular advertisement is chosen. When methods of the Ad Rotator component are called in an ASP page, the component uses this file to select an advertisement to display.

The rotation schedule file is divided into two sections that are separated by an asterisk (*). The first section provides information common to all the ads, and the second section lists specific data for each ad. To test the rotation schedule file, you will use some images from Microsoft.com for your ad images. The following list outlines the structure of the rotation schedule file:

Section 1

- **Redirection**   In URL form, the path and name of the ASP file that can be executed before showing an advertisement. This file can be used to record information about the user who clicks on your ad. You can record information such as the client's IP address, what page the client saw the advertisement on, how often an advertisement was clicked on, and so forth. This ASP file can also handle the case where there is no URL associated with any advertisement in Section 2. When charging advertisers for each hit on their advertisement, it is good practice to prove to them that all the hits aren't resulting from the same user repeatedly hitting **Refresh**.

- **Width** The width of each ad image, in pixels. The default is 440.
- **Height** The height of each ad image, in pixels. The default is 60.
- **Border** The border thickness around each ad image. The default is 1.
- **Asterisk (*)** Separates the first section from the second section. This character must be on a line by itself.

Section 2

You need to complete the following for each advertisement:

- **Image URL** The virtual path and filename of the image file for the advertisement.
- **Advertiser's Home URL** The URL to jump to when this link is selected. If there is no link, use a hyphen (-).
- **Text** The text to display if the browser does not support graphics.
- **Impressions** An integer indicating the relative weight to give to this ad when the Ad Rotator component selects an advertisement. For example, if you list two advertisements, an ad given an impression of 3 has a 30% probability of being selected while an ad given an impression of 7 has a 70% probability of being selected. In this example, the Ad Rotator component selects the Microsoft Windows® advertisement two times out of five and the Microsoft Office advertisement is selected three times out of five.

Copy and paste the following code in your text editor, and save the file as **MyAdRot.txt** in the *x*:\Inetpub\Wwwroot\Tutorial directory.

```
REDIRECT AdRotRedirect.asp
WIDTH 250
HEIGHT 60
BORDER 0
*
http://www.microsoft.com/windows/images/bnrWinfam.gif
http://www.microsoft.com/windows
Microsoft Windows
2
http://www.microsoft.com/office/images/office_logo.gif
http://www.microsoft.com/office
Office 2000
3
```

Example 2: Create an Ad Rotator Redirection File

When a user clicks on an advertisement, an Ad Rotator redirection file written in ASP can capture some information before showing the advertisement and write that information to a file.

For this to work, the *x*:\InetPub\Wwwroot\Tutorial folder must give Read/Write access to the IUSR_*ComputerName* and IWAM_*ComputerName* accounts. Alternatively, you can write this information to a Microsoft Access database using the code from Lesson 3 in Module 1 of this tutorial.

Copy and paste the following code in your text editor, and save the file as **AdRotRedirect.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory.

```
<%@ Language=VBScript %>

<html>
<head>
<title>AdRotRedirect file</title>
</head>
<body>

<%
 'Create some variables.
 dim strLogFile

 'Get the physical path of this Web directory so that we know the path exists.
 'The ASP Server object has many useful methods.
 strLogFile = Server.MapPath(".") & "\AdRotLog.txt"

 'Set some constants for working with files.
 Const cForAppending = 8
 Const cTristateUseDefault = -2

 'Create a FileSystemObject object,
 ' which gives you access to files and folders on the system.
 Set fsoObject = Server.CreateObject("Scripting.FileSystemObject")

 'Open a handle to the file.
 'True means that the file will be created if it doesn't already exist.
 Set tsObject = fsoObject.OpenTextFile(strLogFile, cForAppending, True)

 'Record the data for the user who has just clicked on an advertisement.
 'We have used the Write method of the ASP Request object.
 'The ServerVariables collection of the ASP Request object holds vast
 ' amounts of data for each request made to a Web server.
 tsObject.WriteLine "--------------------"
 tsObject.WriteLine Date & ", " & Time
 tsObject.WriteLine Request.ServerVariables("LOGON_USER")
 tsObject.WriteLine Request.ServerVariables("REMOTE_ADDR")
 tsObject.WriteLine Request.QueryString("url")
 tsObject.WriteLine Request.ServerVariables("HTTP_REFERER")
 tsObject.WriteLine Request.ServerVariables("HTTP_USER_AGENT")
 tsObject.Close

 'Redirect to the Advertiser's Web site using the Redirect method
 ' of the ASP Response object.
 'When the AdRotator component calls AdRotRedirect.asp, it
 ' automatically passes in the advertiser's URL in the QueryString.
 Response.Redirect Request.QueryString("url")
%>

</body>
</html>
```

Example 3: Create an Ad Rotator Include File

Include files are used to store any code that will be used by more than one ASP or HTML file. It makes sense to put your Ad Rotator code into a simple function in an include file. With an Ad Rotator include file, you need to make only one function call from any ASP or HTML file when you want to display an advertisement. Alternatively, you can put the code from the include file in every ASP file where you plan to show an advertisement. However, if you want to change that code, you have to make the change in every ASP file instead of in one include file.

In this example, you create an Ad Rotator include file containing a function named **GetAd**. This function randomly selects ads to display on your ASP pages.

Copy and paste the following code in your text editor, and save the file as **AdRotatorLogic.inc** in the *x*:\Inetpub\Wwwroot\Tutorial directory.

```
<%
 Function GetAd()

   dim objLoad

   'Create an instance of the AdRotator component.
   Set objLoad = Server.CreateObject("MSWC.AdRotator")

   'Set the TargetFrame property, if any. If you have a Web
   ' page using frames, this is the frame where the URL opens.
   'If the HTML page does not find the TARGET name,
   ' the URL will be opened in a new window.
   objLoad.TargetFrame = "TARGET=new"

   'Set one of the other AdRotator properties.
   objLoad.Border = 1

   'Get a random advertisement from the text file.
   GetAd = objLoad.GetAdvertisement("MyAdRot.txt")

 End Function
%>
```

Example 4: Test the Ad Rotator

To test the application you have built on the Ad Rotator component, you need an ASP page that calls the function in the Ad Rotator include file you created.

Copy and paste the following code in your text editor, and save the file as **DisplayAds.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/DisplayAds.asp** in the address bar.

```
<%@ Language=VBScript %>

<html>
<head>
<title>Display an Advertisement</title>
</head>
<body>

<font face="MS Gothic">
<h2>Display an Advertisement</h2>

<comment>Include the file you created to get an advertisement.</comment>
<!-- #include File = "AdRotatorLogic.inc" -->

<comment>Call the Function in the include file.</comment>
<%=GetAd()%>

</font>
</body>
</html>
```

In the browser, you should see the following:

Display an Advertisement

Click the **Refresh** button in your browser about 20 times to watch the advertisement change. Click the advertisement to see how AdRotRedirect.asp redirects you to the advertiser's Web site. Open AdRotLog.txt to see what was recorded when you clicked on an advertisement.

Lesson 2: Counting Page Hits

It may be useful to know how many times someone requests, or *hits*, your Web pages. Sites with high traffic may attract more advertising revenue for you. Some Web sites use this data to charge advertisers a flat fee per hit. Traffic information also indicates how customers are navigating through your site and where ads should be placed. And pages that never seem to be hit might indicate that design changes are needed.

The PageCounter component uses an internal object to record Web page hits on the server. At regular intervals, PageCounter saves all information to a text file so that no counts are lost due to power loss or system failure. The PageCounter component uses three methods, as follows:

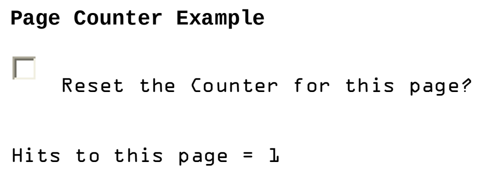
- **Hits** This method displays the number of hits for a Web page. The default is the calling page.
- **PageHit** This method increments the hit count for the current page. If you want hits recorded for an ASP page, this method must be called inside that page.
- **Reset** This method sets the hit count for a page to zero. The default is the calling page.

Copy and paste the following code in your text editor, and save the file as **PageCounter.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/PageCounter.asp** in the address bar.

```
<%@ Language=VBScript %>

<html>
<head>
<title>Page Counter Example</title>
</head>
<body>
<font face="MS Gothic">

<H3>Page Counter Example</H3>
```

```
<p>
<FORM NAME="PageCounter" METHOD="GET" ACTION="PageCounter.asp">
<INPUT TYPE="CHECKBOX" NAME="reset" VALUE="True">Reset the Counter for this page?<BR>
<INPUT TYPE="SUBMIT" VALUE="Submit">
</FORM>
</p>

<%
 'Instantiate the PageCounter object.
 Set MyPageCounter = Server.CreateObject("MSWC.PageCounter")

 'Increment the counter for this page.
 MyPageCounter.PageHit

 If Request.QueryString("reset") = "True" Then
   'Reset the counter for this page.
   MyPageCounter.Reset("/Tutorial/PageCounter.asp")
 End If
%>

Hits to this page = <%=MyPageCounter.Hits %><BR>

</font>
</body>
</html>
```

In the browser, you should see the following:



Page Counter Example

☐ Reset the Counter for this page?

Hits to this page = 1

Click the **Refresh** button in your browser or the **Submit** button on the page to watch the hit count grow. Check the box if you want to reset the counter.

Lesson 3: Creating a Visual Basic COM Object

In this lesson, you use Visual Basic to create a simple COM object, which you can call from an ASP page. This example requires Visual Basic with the ActiveX Wizard, and it is not supported on 64-bit platforms until the Visual Basic runtime is developed for 64-bit platforms. You create a 32-bit COM object that runs on a 64-bit platform, but you must call the 32-bit COM object from a 32-bit application. Because IIS is a 64-bit application on 64-bit platforms, it cannot call 32-bit objects.

Suppose you want to create a Web application that needs functionality VBScript does not have. In this case, you must create a custom procedure that is called, as needed, from any ASP page in your application.

Normally, this approach is an adequate solution for encapsulating custom functionality. However, imagine that you are creating a Web application intended to service thousands of users and that your procedure encapsulates proprietary functions you do not want other people to see. In this case, encapsulating your functionality in the form of a COM component is a superior approach. Components provide better security and performance than scripts because they are compiled code. Components also allow you to use functionality provided by Visual Basic, C++, Java, or any of the other COM-compliant languages.

Create the ActiveX COM Object

The ActiveX DLL Wizard of Visual Basic is the easiest way to create a COM component. You can also use Microsoft Visual C++ to create a COM component, either by using the Active Template Library (ATL) or by writing all the code yourself. This example uses Visual Basic.

In this lesson, you learn how to create and encapsulate a Visual Basic function as a component. Visual Basic includes many financial functions not available to VBScript. This example computes the future value of an investment based on a fixed interest rate and periodic, fixed payments.

1. Open Visual Basic. If you don't see a window titled **New Project**, choose **File** and then click **New Project**.
2. Select **ActiveX DLL**, and click **OK**.
3. A window should open called **Project1 - Class1 (Code)**. This is where you enter your code.

4. From the **Project** menu, click **Project1 Properties**. In the **General** property sheet, in the **Project Name** box, type **ASPTut**. Your DLL is called ASPTut.dll. Select the **Unattended Execution** check box so that the project runs without user interaction and has no user interface elements. Make sure the **Threading Model** is **Apartment Threaded** so that more than one user at a time can use the DLL. Click **OK**.

5. In Visual Basic, you define a class to group together methods and properties. Under the **Project - ASPTut** window, click the **Class1 (Class1)** node to list the class properties below. Under **Properties - Class1**, click in the text field beside **(Name)** and change the class name to **Finance**. When you call this COM component in an ASP page or other script, you will reference it with **ASPTut.Finance**. Click the drop-down box beside **Instancing**, and select **5 - MultiUse**.

6. Learn about the Visual Basic function you are about to use. **FV** is documented on MSDN under the Visual Basic library.

7. The window that was previously titled **Project1 - Class1 (Code)** should now be titled **ASPTut - Finance (Code)**. Copy and paste the following text into that window:

```
Option Explicit

'Declare the global variables that will be set by the Property functions.
Dim gAnnualIntRate As Double
Dim gNumPayPeriods As Integer
Dim gPayment As Double
Dim gPresentSavings As Variant 'Optional
Dim gWhenDue As Variant 'Optional

Public Function CalcFutureValue() As Double

  'The global variables you pass to the FV function are set
  'when user sets the properties in the ASP page.
  'You could pass variables into the CalcFutureValue() function
  'if you wanted to avoid using properties.
  'CalcFutureValue becomes a method in your component.

  Dim IntRatePerPeriod As Double
  Dim FullFutureValue As Double

  If (gAnnualIntRate = Null) Or (gNumPayPeriods = Null) Or (gPayment = Null) Then
    CalcFutureValue = 0
  Else
    IntRatePerPeriod = gAnnualIntRate / 100 / 12
    FullFutureValue = FV(IntRatePerPeriod, gNumPayPeriods, gPayment,
gPresentSavings, gWhenDue)
    CalcFutureValue = Round(FullFutureValue, 2)
  End If

End Function

Public Property Get AnnualIntRate() As Double
  'Get functions return the value of the global variables
  'as if they were properties.
  'In your ASP page, you might say x = oASPTut.Rate.
  AnnualIntRate = gAnnualIntRate
End Property

Public Property Let AnnualIntRate(ByVal vAnnualIntRate As Double)
  'Let functions set the global variables when your ASP page
  'makes a call such as oASPTut.Rate = 5.
  gAnnualIntRate = vAnnualIntRate
End Property

Public Property Get NumPayPeriods() As Integer
  NumPayPeriods = gNumPayPeriods
End Property

Public Property Let NumPayPeriods(ByVal vNumPayPeriods As Integer)
  gNumPayPeriods = vNumPayPeriods
End Property

Public Property Get Payment() As Double
  Payment = gPayment
End Property

Public Property Let Payment(ByVal vPayment As Double)
  gPayment = -(vPayment)
End Property

Public Property Get PresentSavings() As Variant
  PresentSavings = gPresentSavings
End Property

Public Property Let PresentSavings(ByVal vPresentSavings As Variant)
  gPresentSavings = -(vPresentSavings)
```

```
    End Property

    Public Property Get WhenDue() As Variant
      WhenDue = gWhenDue
    End Property

    Public Property Let WhenDue(ByVal vWhenDue As Variant)
      gWhenDue = vWhenDue
    End Property
```

8. All server components require an entry (starting) point. This is the code that will be called when the object is first instantiated with **Server.CreateObject** Your ASPTut component does not have to do anything special when it is first called. For this reason, you can provide an empty **Sub Main** procedure. From the **Project** menu, select **Add Module**. In the **Add Module** window, under the **New** tab, select the **Module** icon and click **Open**. In the **Module 1** code window, type **Sub Main** and hit **Enter**. An empty sub is created for you.

9. Save your **Sub Main** module as **Main.bas**. Save your class file as **Finance.cls**. Save your project as **ASPTut.vbp**.

10. Click **File**, and click **Make ASPTut.dll**. This compiles and registers the ASPTut.dll. After you have called ASPTut.dll from an ASP page, you cannot make the DLL in Visual Basic until you unload the application in which the ASP file is running. One way to do this is to use the IIS snap-in to open the properties on your default Web site and click the **Unload** button. If you want to register your DLL on another Web server, copy ASPTut.dll to the server, click **Start**, click **Run**, and type **cmd** into the **Open** text box. In the same directory as ASPTut.dll, type **regsvr32 ASPTut.dll**.

11. Exit Visual Basic.

Create an ASP Page to Use Your Visual Basic COM Object

This example ASP page uses a form to read in user data, creates an instance of your object, and calculates the future value of your savings plan.

Copy and paste the following code in your text editor, and save the file as **CalculateFutureValue.asp** in the *x*:\Inetpub\Wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/CalculateFutureValue.asp** in the address bar.

```
  <%@ Language=VBScript %>

  <%
  Response.Expires = 0
  Payment = Request.Form("Payment")
  AnnualIntRate = Request.Form("AnnualIntRate")
  NumPayPeriods = Request.Form("NumPayPeriods")
  WhenDue = Request.Form("WhenDue")
  PresentSavings = Request.Form("PresentSavings")
  %>

  <HTML>
  <HEAD><TITLE>Future Value Calculation</TITLE></HEAD>
  <BODY>
  <FONT FACE="MS Gothic">

  <H2 align=center>Calculate the Future Value of a Savings Plan</H2>

  <FORM METHOD=POST ACTION="calculatefuturevalue.asp">
  <TABLE cellpadding=4 align=center>
  <TR>
  <TD>How much do you plan to save each month?</TD>
  <TD><INPUT TYPE=TEXT NAME=Payment VALUE=<%=Payment%>> (Required)</TD>
  </TR><TR>
  <TD>Enter the annual interest rate.</TD>
  <TD><INPUT TYPE=TEXT NAME=AnnualIntRate VALUE=<%=AnnualIntRate%>> (Required)</TD>
  </TR><TR>
  <TD>For how many months will you save?</TD>
  <TD><INPUT TYPE=TEXT NAME=NumPayPeriods VALUE=<%=NumPayPeriods%>> (Required)</TD>
  </TR><TR>
  <TD>When do you make payments in the month? </TD>
  <TD><INPUT TYPE=RADIO NAME=WhenDue VALUE=1 <%If 1=WhenDue Then
Response.Write"CHECKED"%>>Beginning
  <INPUT TYPE=RADIO NAME=WhenDue VALUE=0 <%If 0=WhenDue Then
Response.Write"CHECKED"%>>End </TD>
  </TR><TR>
  <TD>How much is in this savings account now?</TD>
  <TD><INPUT TYPE=TEXT NAME=PresentSavings VALUE=<%=PresentSavings%>> </TD>
  </TR>
  </TABLE>
  <P align=center><INPUT TYPE=SUBMIT VALUE=" Calculate Future Value ">
  </FORM>

  <%
  If ("" = Payment) Or ("" = AnnualIntRate) Or ("" = NumPayPeriods) Then

    Response.Write "<H3 align=center>No valid input entered yet.</H3>"

  ElseIf (IsNumeric(Payment)) And (IsNumeric(AnnualIntRate)) And
(IsNumeric(NumPayPeriods)) Then

    Dim FutureValue
    Set oASPTut = Server.CreateObject("ASPTut.Finance")
    oASPTut.AnnualIntRate = CDbl(AnnualIntRate)
    oASPTut.NumPayPeriods = CInt(NumPayPeriods)
    oASPTut.Payment = CDbl(Payment)
    If Not "" = PresentSavings Then oASPTut.PresentSavings = CDbl(PresentSavings)
    oASPTut.WhenDue = WhenDue
    FutureValue = oASPTut.CalcFutureValue
    Response.Write "<H3 align=center>Future value = $" & FutureValue & "</H3>"

  Else

    Response.Write "<H3 align=center>Some of your values are not numbers.</H3>"

  End If

  %>

  </FONT>
  </BODY>
  </HTML>
```

In the browser, you should see the following:

Calculate the Future Value of a Savings Plan

How much do you plan to save each month?		(Required)
Enter the annual interest rate.		(Required)
For how many months will you save?		(Required)
When do you make payments in the month?	Beginning [✓ End]	
How much is in this savings account now?		

No valid input entered yet.

Lesson 4: Creating a Java COM Object

In this lesson, you use Microsoft® Visual J++® to create a COM object which does the same thing as the Visual Basic component in Lesson 3. This step requires Visual J++ 6.0 or later.

Create the Java COM Object

1. Open Visual J++. If you don't see a window titled **New Project**, click the **File** menu and click **New Project**.
2. Select **Visual J++ Projects**, and click the **Empty Project** icon. In the **Name** text box, type **ASPTut**. Click **Open**.
3. In the **Project** menu, click **Add Class**. In the **Name** text box, type **ASPTut.java**. The class name must be the same as the project name for a Java server component. Click **Open**. The

following should appear in a text editing window:

```
public class ASPTut
{
}
```

4. Copy the following code, and paste it between the brackets {}. Watch capitalization because Java is case-sensitive. The following is a method in your component:

```
public double CalcFutureValue(
  double dblAnnualIntRate,
  double dblNumPayPeriods,
  double dblPayment,
  double dblPresentSavings,
  boolean bWhenDue)
{
  double dblRet, dblTemp, dblTemp2, dblTemp3, dblIntRate;

  if (dblAnnualIntRate == 0.0)
  {
    dblRet = -dblPresentSavings - dblPayment * dblNumPayPeriods;
  }
  else
  {
    dblIntRate = dblAnnualIntRate / 100 / 12;
    dblPayment = -dblPayment;
    dblPresentSavings = -dblPresentSavings;

    dblTemp = (bWhenDue ? 1.0 + dblIntRate : 1.0);
    dblTemp3 = 1.0 + dblIntRate;
    dblTemp2 = Math.pow(dblTemp3, dblNumPayPeriods);
    dblRet = -dblPresentSavings * dblTemp2 - dblPayment * dblTemp * (dblTemp2 - 1.0)
/ dblIntRate;
  }

  return dblRet;
}
```

5. From the **Build** menu, click **Build**. Look in the **Task List** window below the text editing window to see whether any errors are generated.

6. The Java class file must be registered on the same machine as the Web server. In a command window, find the ASPTut.class file that was built. It is most likely either in %USERPROFILE%\My Documents\Visual Studio Projects\ASPTut or in *x*:\Documents and Settings*user name*\My Documents\Visual Studio Projects\ASPTut, where x: is the drive on which you installed Windows. Copy ASPTut.class to *x*:\Winnt\Java\Trustlib. Type **javareg /register /class:ASPTut /progid:MS.ASPTut.Java**, and press **ENTER** to register the Java class.

7. Close Visual J++.

Create an ASP Page to Use Your Java COM Object

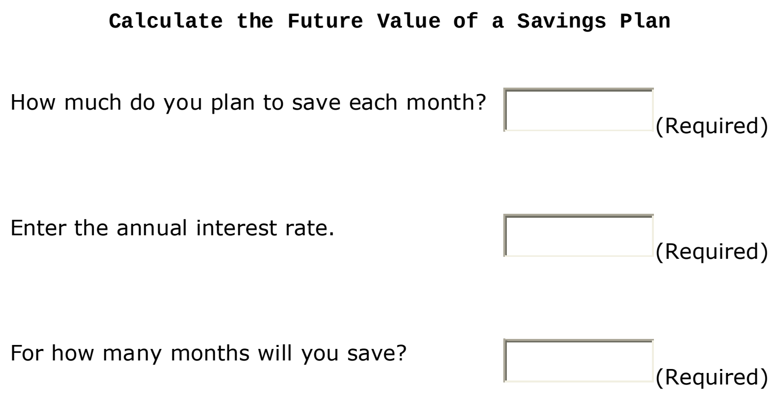
This example ASP page uses a form to read in user data, creates an instance of your object, and calculates the future value of your savings plan. This example uses JScript, but you can call a Java component from VBScript as well.

Copy and paste the following code in your text editor, and save the file as **CalculateFutureValueJava.asp** in the *x*:\Inetpub\wwwroot\Tutorial directory. View the example with your browser by typing **http://localhost/Tutorial/CalculateFutureValueJava.asp** in the address bar.

```
<%@ Language=JScript %>

<%
Response.Expires = 0;
Payment = Request.Form("Payment");
AnnualIntRate = Request.Form("AnnualIntRate");
NumPayPeriods = Request.Form("NumPayPeriods");
WhenDue = Request.Form("WhenDue");
PresentSavings = Request.Form("PresentSavings");
%>

<HTML>
<HEAD><TITLE>Future Value Calculation - Java</TITLE></HEAD>
<BODY>
<FONT FACE="MS Gothic">

<H2 align=center>Calculate the Future Value of a Savings Plan</H2>

<FORM METHOD=POST ACTION="calculatefuturevaluejava.asp">
<TABLE cellpadding=4 align=center>
```

```
  <TR>
  <TD>How much do you plan to save each month?</TD>
  <TD><INPUT TYPE=TEXT NAME=Payment VALUE=<%=Payment%>> (Required)</TD>
  </TR><TR>
  <TD>Enter the annual interest rate.</TD>
  <TD><INPUT TYPE=TEXT NAME=AnnualIntRate VALUE=<%=AnnualIntRate%>> (Required)</TD>
  </TR><TR>
  <TD>For how many months will you save?</TD>
  <TD><INPUT TYPE=TEXT NAME=NumPayPeriods VALUE=<%=NumPayPeriods%>> (Required)</TD>
  </TR><TR>
  <TD>When do you make payments in the month? </TD>
  <TD><INPUT TYPE=RADIO NAME=WhenDue VALUE=1 <%if (1==WhenDue)
Response.Write("CHECKED")%>>Beginning
  <INPUT TYPE=RADIO NAME=WhenDue VALUE=0 <%if (0==WhenDue)
Response.Write("CHECKED")%>>End </TD>
  </TR><TR>
  <TD>How much is in this savings account now?</TD>
  <TD><INPUT TYPE=TEXT NAME=PresentSavings VALUE=<%=PresentSavings%>> </TD>
  </TR>
  </TABLE>
  <P align=center><INPUT TYPE=SUBMIT VALUE=" Calculate Future Value ">
  </FORM>

  <%

  if (("" == Payment) || ("" == AnnualIntRate) || ("" == NumPayPeriods)) {

    Response.Write("<H3 align=center>No valid input entered yet.</H3>");

  } else {

    AnnualIntRate = parseFloat(AnnualIntRate)
    NumPayPeriods = parseFloat(NumPayPeriods)
    Payment = parseFloat(Payment)
    if ("" != PresentSavings) PresentSavings = parseFloat(PresentSavings);

    if ((isNaN(Payment)) || (isNaN(AnnualIntRate)) || (isNaN(NumPayPeriods))) {

      Response.Write("<H3 align=center>Some of your values are not numbers.</H3>");

    } else {

      var FutureValue, Cents;
      var oASPTut = Server.CreateObject("MS.ASPTut.Java");
      FutureValue = oASPTut.CalcFutureValue(AnnualIntRate, NumPayPeriods, Payment,
PresentSavings, WhenDue);

      Response.Write("<H3 align=center>Future value = $" + parseInt(FutureValue) +
"</H3>");

    }
  }
  %>

  </FONT>
  </BODY>
  </HTML>
```

In the browser, you should see the following content, which should be identical to the display generated using the Visual Basic component in Lesson 3 of this module:



Calculate the Future Value of a Savings Plan

How much do you plan to save each month? (Required)

Enter the annual interest rate. (Required)

For how many months will you save? (Required)

When do you make payments in the month? Beginning [✓ End]

How much is in this savings account now?

No valid input entered yet.

Up Next: Maintaining Session State in a Web Application

Module 3: Maintaining Session State

This module describes the process of maintaining session state in Active Server Pages (ASP). Session refers to the time segment that a specific user is actively viewing the contents of a Web site. A session starts when the user visits the first page on the site, and it ends a few minutes after the user leaves the site. The pieces of user-specific information, relevant to a particular session, are collectively known as *session state*.

Because HTTP is a stateless protocol, a problem arises when trying to maintain state for a user visiting your Web site. The Web server treats each HTTP request as a unique request, which is unrelated to any previous requests. Thus, the information a user enters on one page (through a form, for example) is not automatically available on the next page requested. The Web server must maintain session state to identify and track users as they browse through pages on a Web site.

One solution is through the use of *cookies*. Cookies record information about a user on one page and transfer that information to other pages within the site. However, a few browsers do not recognize cookies, and on other browsers, users can disable cookies. If you are concerned about reaching this Web audience, you can maintain session state without using cookies by using HTTP POST.

This module includes the following lessons:

- Maintaining Session State with Cookies   Provides two cookie examples, one using the ASP Response and Request objects and another using the ASP Session object.
- Maintaining Session State Without Cookies   Provides an example of the alternative to maintaining session state with cookies: HTTP POST.

Maintaining Session State with Cookies

Cookies store a set of user specific information, such as a credit card number or a password. The Web server embeds the cookie into a user's Web browser so that the user's information becomes available to other pages within the site; users do not have to reenter their information for every page they visit. Cookies are a good way to gather customer information for Web-based shopping, for retaining the personal preferences of the Web user, or for maintaining state about the user.

There are two kinds of cookies, as follows:

- **In-memory cookies**   An in-memory cookie goes away when the user shuts the browser down.
- **Persistent cookies**   A persistent cookie resides on the hard drive of the user and is retrieved when the user comes back to the Web page.

If you create a cookie without specifying an expiration date, you are creating an in-memory cookie, which lives for that browser session only. The following illustrates the script that would be used for an in-memory cookie:

```
Response.Cookies("SiteArea") = "TechNet"
```

If you want the cookie information to persist beyond the session, you should create a persistent cookie by specifying an expiration date. Supplying an expiration date causes the browser to save the cookie on the client computer. Until the cookie expiration date is reached, the data in the persistent cookie will stay on the client machine. Any request to the original Web site will automatically attach the cookie that was created by that site. Cookies go only to the sites that created them because part of the Web site name and ASP file make up the data in the cookie.
The following illustrates the script used to create a persistent cookie:

```
Response.Cookies("SiteArea") = "TechNet"
Response.Cookies("SiteArea").Expires = "August 15, 2000"
```

The script to create a cookie should be placed at the beginning of the ASP file because cookies need to be generated before any HTML text is sent to the browser.

Cookies Using the Response and Request Objects

Persistent cookies are produced using the **Response** and **Request** objects, although these objects may also be used to create an in-memory cookie. The majority of Web applications employ these objects to maintain session state.

- **Response object**   Use the **Response** object to create and set cookie values.

- **Request object** Use the **Req**uest object to retrieve the value of a cookie created during a previous Web session.

In this lesson you will use the **Response** and **Request** objects to create the following files. Please create them all at once, because some of them need the others. After you have created all the files, run the application by typing **http://LocalHost/Tutorial/Frame.htm** in your browser.

- **Frame.htm** A page that splits the the user's view into two windows. This page requires that Menu.htm and CustomGreeting.asp.
- **Menu.htm** A page containing links to the samples for this lesson. For the links to work, this page requires that all the other pages have been created.
- **CustomGreeting.asp** An ASP script that takes the user's name in a form and sets an in-memory cookie.
- **DeleteGreetingCookie.asp** An ASP script that deletes the cookie that contains the user's name. If no cookie is set, a warning is displayed.
- **SelectColors.asp** An ASP script that sets up the cookies for the user's color choices.
- **DeleteColorCookie.asp** An ASP script that deletes the Web colors previously chosen. If none are chosen, a warning is displayed.
- **Cookie.asp** An ASP script that sets persistent cookies to hold the current date and time of the user's visit and record the total number of visits.
- **DeleteCookies.asp** This ASP script deletes the cookies set in Cookie.asp. If no cookies are set, a warning is displayed.

Frame.htm

Open a new file in your text editor, paste in the following script, and save the file as **C:\Inetpub\Wwwroot\Tutorial\Frame.htm**.

```
  <html>
  <head>
  <title>Customized Greeting and Colors Using In-Memory and Persistent Cookies</title>
  </head>

  <frameset cols="40%,60%">
    <frame src="menu.htm" name="left" marginheight="5" marginwidth="5">
    <frame src="CustomGreeting.asp" name="right" marginheight="5" marginwidth="5">
  </frameset>

  <noframes>
    Sorry, your browser does not support frames. Please go to the <a
href="menu.htm">Menu</a>.
  </noframes>

  </html>
```

Menu.htm

Open a new file in your text editor, paste in the following script, and save the file as **C:\Inetpub\Wwwroot\Tutorial\Menu.htm**.

```
  <html>
  <head>
  <title>Maintaining Session State With Cookies</title>
  </head>
  <body>
  <font face="MS Gothic">

  <h2 align="center">Cookie Examples</h2>

  <table align=center border=1 cellpadding=4>
    <tr>
    <td><a href="CustomGreeting.asp" target="right"><b>Custom Greeting Page</b></a></td>
    </tr><tr>
    <td><a href="DeleteGreetingCookie.asp" target="right"><b>Delete the Greetings
Cookie</b></a></td>
    </tr><tr>
    <td><a href="SelectColors.asp" target="right"><b>Set Page Colors</b></a></td>
    </tr><tr>
    <td><a href="DeleteColorCookie.asp" target="right"><b>Delete Page Colors
Cookies</b></a></td>
    </tr><tr>
    <td><a href="Cookie.asp" target="right"><b>Set Cookies for Date, Time and Total
```

```
Visits</b></a></td>
    </tr><tr>
    <td><a href="DeleteCookies.asp" target="right"><b>Delete Cookies for Date, Time and
Total Visits</b></a></td>
    </tr>
  </table>

  </font>
  </body>
  </html>
```

CustomGreeting.asp

Open a new file in your text editor, paste in the following script, and save the file as **C:\Inetpub\Wwwroot\Tutorial\CustomGreeting.asp**.

```
  <%@ Language="VBScript" %>
  <%
   'If the user has selected text and background colors,
   ' cookies are used to remember the values between HTTP sessions.
   'Do this first so that your page can use use the values if they are set.
   If Not (Request.QueryString("Text")="") Then
     Response.Cookies("TextColor") = Request.QueryString("Text")
     Response.Cookies("BackgroundColor") = Request.QueryString("Background")
   End If

   ' If the user has typed in a name, a cookie is created.
   If Not (Request.QueryString("Name")="") Then
     Response.Cookies ("Name") = Request.QueryString("Name")

   ' If the user does not give his/her name, a cookie
   ' is created so that we do not keep asking for the name.
   ElseIf (InStr(Request.QueryString,"Name")=1) Then
     Response.Cookies ("NoUserInput") = "TRUE"

   End If
  %>

  <html>
  <head>
  </head>

  <%
   'Set colors according to existing previous user input.
   If (Request.Cookies ("TextColor")="") Then %>
     <body>
   <% Else %>
     <body bgcolor=<%=Request.Cookies("BackgroundColor")%>
text=<%=Request.Cookies("TextColor")%>>
   <% End If
  %>

  <font face="MS Gothic">

  <%
   'If there is no name cookie set, no name entered by the user,
   ' and there was no user input at all, get the user's name.
   If ( (Request.Cookies("Name")="") And ((Request.QueryString("Name"))="")) And
(Not(Request.Cookies("NoUserInput")="TRUE") ) Then %>

      <FORM ACTION="CustomGreeting.asp" METHOD="GET" NAME="DataForm">
     <table align=center><tr><td>
     <INPUT TYPE=TEXTBOX NAME="Name" SIZE=33></td></tr><tr><td>
     <INPUT TYPE=Submit VALUE="Please Enter Your Name"></td></tr></table>
     </FORM>

   <% ElseIf Not(Request.Cookies("Name")="") Then %>

     <H2 align=center>Greetings <%=Request.Cookies("Name")%></H2>

   <% Else %>

     <H2>Hello!</H2>
     <H3>You did not give us your name so we are not able to greet you by name.</H3>

   <% End If
  %>

  <H3>In-Memory Cookie Example</H3>
  <P>
  Once you enter your name:
  <UL>
  <LI>If you hit <B>Refresh</B> in your browser, you should still see your name.</LI>
  <LI>If you close your browser, the cookie is deleted. When you re-open your browser to
this page, you should be asked for your name again.</LI>
  <LI>If you click on <B>Delete the Greetings Cookie</B>, and click on <B>Custom Greeting
```

```
Page</B>, you should be asked for your name again.</LI>
  </P>

  </font>
  </body>
  </html>
```

DeleteGreetingCookie.asp

Open a new file in your text editor, paste in the following script, and save the file as **C:\Inetpub\Wwwroot\Tutorial\DeleteGreetingCookie.asp**.

```
  <%@ Language="VBScript" %>

  <html>
  <head>
  </head>

  <% If (Request.Cookies ("TextColor")="") Then %>
     <body>
     <font face="MS Gothic">
  <% Else %>
     <body bgcolor=<%=Request.Cookies("BackgroundColor")%>
text=<%=Request.Cookies("TextColor")%>>
     <font face="MS Gothic" color=<%=Request.Cookies("TextColor")%>>
  <% End If %>

  <%
   If Not ("" = Request.Cookies("Name")) Then
     Response.Cookies ("Name").Expires = "January 1, 1992"
     Response.Cookies ("NoUserInput").Expires = "January 1, 1992" %>

     <h2 align=center>In-Memory Greeting Cookie Deleted</h2>
     <P>
     The cookie used to keep track of your name has been deleted.<BR>
     Please click on <B>Custom Greeting Page</B> to be asked for your name again.
     </P>

   <% Else %>

     <h2 align=center>No In-Memory Greeting Cookie Deleted</h2>
     <P>
     There was no cookie set with your name.<BR>
     Please click on <B>Custom Greeting Page</B> to enter your name.
     </P>

   <% End If
  %>

  </font>
  </body>
  </html>
```

SelectColors.asp

Open a new file in your text editor, paste in the following script, and save the file as **C:\Inetpub\Wwwroot\Tutorial\SelectColors.asp**.

```
  <%@ Language="VBScript" %>

  <%
    ' If the user has selected text and background colors,
    ' cookies are used to remember the values between HTTP sessions.
    If Not (Request.QueryString("Text")="") Then
      Response.Cookies ("TextColor") = Request.QueryString("Text")
      Response.Cookies ("BackgroundColor") = Request.QueryString("Background")
    End If
  %>

  <html>
  <head>
  </head>

  <%
    'Set colors according to existing previous user input.
    If (Request.Cookies ("TextColor")="") Then %>
     <body>
    <% Else %>
      <body bgcolor=<%=Request.Cookies("BackgroundColor")%>
text=<%=Request.Cookies("TextColor")%>>
    <% End If
  %>

  <font face="MS Gothic">
  <H2 align=center>Select the colors for your Web page</H2>
  <P>
  In Memory Cookies will be used to store these values.
  </P>
  <FORM ACTION="SelectColors.asp" METHOD="GET" NAME="DataForm">
  <table border="1" width="450" cellpadding=0>
  <tr><td>
    <table>
    <tr><td BGCOLOR=99FF99>
    <B><font color=000000>Please select the background color</font></B>
    </td></tr><tr><td BGCOLOR=FFFFFF>
    <input type="RADIO" NAME="Background" VALUE="FFFFFF" CHECKED><font COLOR=000000>
FFFFFF </font>
    </td></tr><tr><td BGCOLOR=D98719>
    <input type="RADIO" NAME="Background" VALUE="D98719"> D98719
    </td></tr><tr><td BGCOLOR=D9D919>
    <input type="RADIO" NAME="Background" VALUE="D9D919"> D9D919
    </td></tr><tr><td BGCOLOR=00FFFF>
    <input type="RADIO" NAME="Background" VALUE="00FFFF"> 00FFFF
    </td></tr><tr><td BGCOLOR=FF00FF>
    <input type="RADIO" NAME="Background" VALUE="FF00FF"> FF00FF
    </td></tr><tr><td BGCOLOR=000000>
    <input type="RADIO" NAME="Background" VALUE="000000"> <font
COLOR=FFFFFF>000000</font>
    </td></tr>
  </table>
  </td><td>
    <table>
    <tr><td BGCOLOR=99FF99>
    <B><font color=000000>Please select the text color</font></B>
    </td></tr><tr><td BGCOLOR=FFFFFF>
    <input type="RADIO" NAME="Text" VALUE="FFFFFF" CHECKED><font COLOR=000000> FFFFFF
</font>
    </td></tr><tr><td BGCOLOR=D98719>
    <input type="RADIO" NAME="Text" VALUE="D98719"> D98719
    </td></tr><tr><td BGCOLOR=D9D919>
    <input type="RADIO" NAME="Text" VALUE="D9D919"> D9D919
    </td></tr><tr><td BGCOLOR=00FFFF>
    <input type="RADIO" NAME="Text" VALUE="00FFFF"> 00FFFF
    </td></tr><tr><td BGCOLOR=FF00FF>
    <input type="RADIO" NAME="Text" VALUE="FF00FF"> FF00FF
    </td></tr><tr><td BGCOLOR=000000>
    <input type="RADIO" NAME="Text" VALUE="000000" CHECKED><font COLOR=FFFFFF> 000000
</font>
    </td></tr>
     </table>
  </td></tr>
  </table>
  <P>
  <input type=Submit VALUE="Submit selected colors">
  </FORM>

  </font>
  </body>
  </html>
```

DeleteColorCookie.asp

Open a new file in your text editor, paste in the following script, and save the file as **C:\Inetpub\Wwwroot\Tutorial\DeleteColorCookie.asp**.

```
  <%@ Language="VBScript" %>

  <html>
  <head>
  </head>
  <body>
  <font face="MS Gothic">

  <%
   If Not ("" = Request.Cookies("TextColor")) Then
     Response.Cookies("TextColor").Expires = "January 1, 1992"
     Response.Cookies("BackgroundColor").Expires = "January 1, 1992" %>

     <h2 align=center>In-Memory Color Cookie Deleted</h2>
     <P>
     The cookie used to keep track of your display colors has been deleted.<BR>
     Please click on <B>Set Page Colors</B> to set your colors again.
     </P>

   <% Else %>

     <h2 align=center>No In-Memory Color Cookie Deleted</h2>
     <P>
     There was no cookie set with your color choices.<BR>
     Please click on <B>Set Page Colors</B> to set display colors.
```

```
    </P>

   <% End If
  %>

  </font>
  </body>
  </html>
```

Cookie.asp

Open a new file in your text editor, paste in the following script, and save the file as **C:\Inetpub\Wwwroot\Tutorial\Cookie.asp**.

```
  <%@ Language="VBScript" %>

  <%
   LastAccessTime = Request.Cookies("LastTime")
   LastAccessDate = Request.Cookies("LastDate")

   'If the NumVisits cookie is empty, set to 0, else increment it.
   If (Request.Cookies("NumVisits")="") Then
     Response.Cookies("NumVisits") = 0
   Else
     Response.Cookies("NumVisits") = Request.Cookies("NumVisits") + 1
   End If

   Response.Cookies("LastDate") = Date
   Response.Cookies("LastTime") = Time

   'Setting an expired date past the present date creates a persistent cookie.
   Response.Cookies("LastDate").Expires = "January 15, 2001"
   Response.Cookies("LastTime").Expires = "January 15, 2001"
   Response.Cookies("NumVisits").Expires = "January 15, 2001"
  %>

  <html>
  <head>
  </head>
  <% If (Request.Cookies ("TextColor")="") Then %>
     <body>
     <font face="MS Gothic">
  <% Else %>
     <body bgcolor=<%=Request.Cookies("BackgroundColor")%>
text=<%=Request.Cookies("TextColor")%>>
     <font face="MS Gothic" color=<%=Request.Cookies("TextColor")%>>
  <% End If %>

  <H2 align=center>Persistent Client-Side Cookies!</H2>

  <P>
  Three persistent client-side cookies are created.
  <UL>
  <LI>A cookie to count the number of times you visited the Web page.</LI>
  <LI>A cookie to determine the date of your visit.</LI>
  <LI>A cookie to determine the time of your visit.</LI>
  </UL>
  </P>

 <table border="1" width="300" cellpadding=4 align=center>
 <tr><td>
 <% If (Request.Cookies ("NumVisits")=0) Then %>
    Welcome! This is your first visit to this Web page!
 <% Else %>
    Thank you for visiting again! You have been to this Web page a total of
<B><%=Request.Cookies("NumVisits")%></B> time(s).
 <% End If %>
 </td></tr>
 </table>

 <P>
 <B>The Current time is <%=Time%> on <%=Date%><BR>
 <% If (Request.Cookies ("NumVisits")>0) Then %>
     You last visited this Web page at <%=LastAccessTime%> on <%=LastAccessDate%>
 <% End If %>
 </strong>
 </P>

 </font>
 </body>
 </html>
```

DeleteCookies.asp

Open a new file in your text editor, paste in the following script, and save the file as DeleteCookies.asp.

```
  <%@ Language="VBScript" %>

  <html>
  <head>
  </head>

  <% If (Request.Cookies ("TextColor")="") Then %>
     <body>
     <font face="MS Gothic">
  <% Else %>
     <body bgcolor=<%=Request.Cookies("BackgroundColor")%>
text=<%=Request.Cookies("TextColor")%>>
     <font face="MS Gothic" color=<%=Request.Cookies("TextColor")%>>
  <% End If %>

  <%
   If Not ("" = Request.Cookies("NumVisits")) Then
     Response.Cookies("NumVisits").Expires = "January 1, 1993"
     Response.Cookies("LastDate").Expires = "January 1, 1993"
     Response.Cookies("LastTime").Expires = "January 1, 1993" %>

     <H2 align=center>Persistent Cookies Are Deleted</H2>
     <P>
     The cookies used to keep track of your visits and date and time of last visit have
been deleted.<BR>
     Please click on <B>Set Cookies for Date, Time and Total Visits</B> to set your
cookies again.
     </P>

   <% Else %>

     <H2 align=center>No Persistent Cookies Are Deleted</H2>
     <P>
     There were no cookies set to keep track of your visits, and date and time of last
visit.<BR>
     Please click on <B>Set Cookies for Date, Time and Total Visits</B> to set your
colors again.
     </P>

   <% End If %>

  </font>
  </body>
  </html>
```

Cookies Using the Session Object

With the **Session** object, you can create only an in-memory cookie. For the **Session** object to work correctly, you need to determine when a user's visit to the site begins and ends. IIS does this by using a cookie that stores an ASP Session ID, which is used to maintain a set of information about a user. If an ASP Session ID is not present, the server considers the current request to be the start of a visit. The visit ends when there have been no user requests for ASP files for the default time period of 20 minutes.

In this lesson, you will create the following:

- **Global.asa** Global.asa is a file that allows you to perform generic actions at the beginning of the application and at the beginning of each user's session. An application starts the first time the first user ever requests a page and ends when the application is unloaded or when the server is taken offline. A unique session starts once for each user and ends 20 minutes after that user has requested their last page. Generic actions you can perform in Global.asa include setting application or session variables, authenticating a user, logging the date and time that a user connected, instantiating COM objects that remain active for an entire application or session, and so forth.
- **VisitCount.asp** This ASP script uses the **Session** object to create an in-memory cookie.

When an application or session begins or ends, it is considered an event. Using the Global.asa file, you can use the predefined event procedures that run in response to the event.

Global.asa

Open a new file in your text editor, paste in the following script, and save the file in your root directory as **C:\Inetpub\Wwwroot\Global.asa**.

Important: Global.asa files must be saved in the root directory of the application for ASP to find it. If you had a virtual path called Test mapped to C:\Inetpub\Wwwroot\Test, your URL would be http://LocalHost/Test, and the Global.asa file would have to go in C:\Inetpub\Wwwroot\Test. We did not create a virtual path mapped to C:\Inetpub\Wwwroot\Tutorial, so our root directory is still C:\Inetpub\Wwwroot.

```
  <SCRIPT LANGUAGE=VBScript RUNAT=Server>

  'Using application-level variables to track the number of users
  ' that are currently looking at the site and the number that have
  ' accessed the site.
  Sub Application_OnStart

    'Get the physical path to this vdir, and append a filename.
    Application("PhysPath") = Server.MapPath(".") & "\hits.txt"

    'Set some Visual Basic constants, and instantiate the FileSystemObject object.
    Const cForReading = 1
    Const cTristateUseDefault = -2
    Set fsoObject = Server.CreateObject("Scripting.FileSystemObject")

    'Get the last saved value of page hits and the date that it happened.
    If fsoObject.FileExists(Application("PhysPath")) Then

      'If the file hits.txt exists, set the Application variables.
      Set tsObject = fsoObject.OpenTextFile(Application("PhysPath"), cForReading,
cTristateUseDefault)
      Application("HitCounter") = tsObject.ReadLine
      Application("AppStartDate") = tsObject.ReadLine
      tsObject.Close

    Else 'No file has been saved, so reset the values.

      Application("HitCounter") = 0
      Application("AppStartDate") = Date

    End If

    Application("CurrentUsers") = 0

  End Sub


  Sub Application_OnEnd

    Const cForWriting = 2
    Const cTristateUseDefault = -2

    Set fsoObject = Server.CreateObject("Scripting.FileSystemObject")
    If fsoObject.FileExists(Application("PhysPath")) Then

      'If the file exists, open it for writing.
      set tsObject = fsoObject.OpenTextFile(Application("PhysPath"), cForWriting,
cTristateUseDefault)

    Else

      'If the file doesn't exist, create a new one.
      set tsObject = fsoObject.CreateTextFile(Application("PhysPath"))

    End If

    'Write the total number of site hits and the last day recorded to the file.
    tsObject.WriteLine(Application("HitCounter"))
    tsObject.WriteLine(Application("AppStartDate"))
    tsObject.Close

  End Sub


  Sub Session_OnStart

    'The Session time-out default is changed to 1 for the purposes of
    ' this example.
    Session.Timeout = 1

    'When you change Application variables, you must lock them so that other
    ' sessions cannot change them at the same time.
    Application.Lock

    'Increment the site hit counter.
    Application("HitCounter") = Application("HitCounter") + 1
    Application("CurrentUsers") = Application("CurrentUsers") + 1

    Application.UnLock

  End Sub


  Sub Session_OnEnd

    Application.Lock
    'Decrement the current user counter.
    Application("CurrentUsers") = Application("CurrentUsers") - 1

    Application.UnLock

  End Sub

  </SCRIPT>
```

VisitCount.asp

You can use variables set in Global.asa to measure visits and sessions.

Open a new file in your text editor, paste in the following script, and save the file as **C:\Inetpub\Wwwroot\Tutorial\VisitCount.asp**. View the file in your browser by typing http://Localhost/Tutorial/VisitCount.asp.

Open a second instance of the browser to http://Localhost/Tutorial/VisitCount.asp, and click **Refresh** on the first browser. Total Visitors and Active Visitors should increase by one. Close down the second browser, wait over a minute, and click **Refresh** on the first browser. Active Visitors should decrease by one.

```
  <% Response.Buffer = True%>

  <html>
  <head>
  <title>Retrieving Variables Set in Global.asa</title>
  </head>
  <body>
  <font face="MS Gothic">

  <H3 align=center>Retrieving Variables Set in Global.asa</H3>
  <P>
  Total Visitors = <%=Application("HitCounter")%> since
<%=Application("AppStartDate")%><BR>
  Active Visitors = <%=Application("CurrentUsers")%>
  </P>

  </font>
  </body>
  </html>
```

Maintaining Session State Without Cookies

Some browsers do not recognize cookies, and users can choose to disable cookies in their browsers. The HTTP POST method provides an alternative to cookies to maintain session state. The HTTP POST method provides the same state information as would a cookie but has the advantage that it works even when cookies are not available. This method is not common in practice, but it is a good example to learn from. The HTTP POST method works similarly to an in-memory cookie; user information can be maintained only during the visit, and the session state information is gone when the user turns off the browser.

DataEntry.asp

Open a new file in your text editor, paste in the following script, and save the files as **C:\Inetpub\Wwwroot\Tutorial\DataEntry.asp**. View the file in your browser by typing http://Localhost/Tutorial/DataEntry.asp.

```
  <%@ Language=VBScript %>

  <html>
  <head>
  <title>Data Entry Without Cookies</title>
  </head>
  <body>
  <font face="MS Gothic">

  <!-- In this example, subroutines are listed first.
       There's a subroutine for each page of the order process.
       The main calling code is at the bottom. -->

  <% Sub DisplayInitialPage %>

    <table border=1 cellpadding=3 cellspacing=0 width=500 bordercolor=#808080
align=center>
    <tr><td bgColor=#004080 align=center>
    <font color=#ffffff><H2>Order Form</H2></font>
    </td></tr><tr><td bgColor=#e1e1e1 align=left>
    <P><B>Step 1 of 4</B></P>
    <P align=center>
    This form uses the HTTP POST method to pass along hidden values that contain
    your order information. This form does not use cookies.
    </P>

    <FORM METHOD=POST ACTION="DataEntry.asp" NAME=DataEntryForm>
    <P>Enter your name
    <INPUT TYPE="TEXT" NAME=FullName>
    <BR>Enter your imaginary credit card number
    <INPUT TYPE="TEXT" NAME=CreditCard>
    </P>
    <!-- Keeps track of the information by using the hidden HTML form variable Next Page.
-->
    <INPUT TYPE="HIDDEN" NAME=NextPage VALUE=2>
    <INPUT TYPE="SUBMIT" VALUE="Next ->" NAME=NextButton>
    </FORM>

    </td></tr>
    </table>

  <% End Sub %>


  <% Sub DisplayDogBreed %>

    <table border=1 cellpadding=3 cellspacing=0 width=500 align=center>
    <tr><td bgColor=#004080 align=center>
    <font color=#ffffff><H2>Order Form</H2></font>
    </td></tr><tr><td bgColor=#e1e1e1>
    <P><B>Step 2 of 4</B></P>
    <P align=center>
    Please select the type of dog you want.
    </P>

    <FORM METHOD=POST ACTION="DataEntry.asp" NAME=DataEntryForm>
    <P>
    <INPUT TYPE=RADIO NAME=DogSelected VALUE="Cocker Spaniel" CHECKED>Cocker Spaniel<BR>
    <INPUT TYPE=RADIO NAME=DogSelected VALUE="Doberman">Doberman<BR>
    <INPUT TYPE=RADIO NAME=DogSelected VALUE="Timber Wolf">Timber Wolf<BR>
    <INPUT TYPE=RADIO NAME=DogSelected VALUE="Mastiff">Mastiff<BR>
    </P>
    <!--Keeps track of the information by using the hidden HTML form variable Next Page.
-->
    <INPUT TYPE="HIDDEN" NAME=NextPage VALUE=3>
    <INPUT TYPE="SUBMIT" VALUE="Next ->" NAME=NextButton>
    </FORM>
    </td></tr>
    </table>

  <% End Sub %>


  <% Sub DisplayCity %>

    <table border=1 cellpadding=3 cellspacing=0 width=500 align=center>
    <tr><td bgColor=#004080 align=center>
    <font color=#ffffff><H2>Order Form</H2></font>
    </td></tr><tr><td bgColor=#e1e1e1>
    <P><B>Step 3 of 4</B></P>
    <P align=center>
    We deliver from the following cities. Please choose the one closest to you.
    </P>

    <FORM METHOD=POST ACTION="DataEntry.asp" NAME=DataEntryForm>
    <P>
    <INPUT TYPE=RADIO NAME=CitySelected VALUE="Seattle" CHECKED>Seattle<BR>
    <INPUT TYPE=RADIO NAME=CitySelected VALUE="Los Angeles">Los Angeles<BR>
    <INPUT TYPE=RADIO NAME=CitySelected VALUE="Boston">Boston<BR>
    <INPUT TYPE=RADIO NAME=CitySelected VALUE="New York">New York<BR>
    </P>
    <!--Keeps track of the information by using the hidden HTML form variable Next Page.
-->
    <INPUT TYPE="HIDDEN" NAME=NextPage VALUE=4>
    <INPUT TYPE="SUBMIT" VALUE="Next ->" NAME=NextButton>
    </FORM>
    </td></tr>
    </table>

  <% End Sub %>


  <% Sub DisplaySummary %>

    <table border=1 cellpadding=3 cellspacing=0 width=500 align=center>
    <tr><td bgColor=#004080 align=center>
    <font color=#ffffff><H2>Order Form Completed</H2></font>
    </td></tr><tr><td bgColor=#e1e1e1>
    <P><B>Step 4 of 4</B></P>
    <P align=center>
    The following information was entered.<BR>
    A transaction will now be executed to complete your order if your name and
    credit card are valid.
    </P>
      <table cellpadding=4>
      <tr bgcolor=#ffffcc><td>
      Name
      </td><td>
      <%=Session.Value("FullName")%>
      </td></tr><tr bgcolor=Beige><td>
      Credit Card
      </td><td>
      <%=Session.Value("CreditCard")%>
      </td></tr><tr bgcolor=Beige><td>
      Dog Ordered
      </td><td>
      <%=Session.Value("DogSelected")%>
      </td></tr><tr bgcolor=Beige><td>
      City Ordered From
      </td><td>
      <%=Session.Value("CitySelected")%>
      </td></tr>
      </table>
    </td>
    </tr>
    </table>

  <% End Sub %>


  <% Sub StoreUserDataInSessionObject %>
  <%
    Dim FormKey
    For Each FormKey in Request.Form
    Session(FormKey) = Request.Form.Item(FormKey)
    Next
  %>
  <% End Sub %>


  <%
    'This is the main code that calls all the subroutines depending on the
    ' hidden form elements.

    Dim CurrentPage

    If Request.Form.Item("NextPage") = "" Then
      CurrentPage = 1
    Else
      CurrentPage = Request.Form.Item("NextPage")
    End If

    'Save all user data so far.
    Call StoreUserDataInSessionObject

    Select Case CurrentPage
      Case 1 : Call DisplayInitialPage
      Case 2 : Call DisplayDogBreed
      Case 3 : Call DisplayCity
      Case 4 : Call DisplaySummary
    End Select %>

  <BR>
  <HR>
  <H3 align=center><A HREF="DataEntry.asp">Reset Order</A></H3>

  </font>
  </body>
  </html>
```

Creating an ASP Page

An Active Server Pages (ASP) file is a text file with the extension .asp that contains any combination of the following:

- Text
- HTML tags
- Server-side scripts

A quick way to create an .asp file is to rename your HTML files by replacing the existing .htm or .html file name extension with an .asp extension. If your file does not contain any ASP functionality, then the server dispenses with the ASP script processing and efficiently sends the file to the client. As a Web developer, this affords you tremendous flexibility because you can assign your files .asp extensions, even if you do not plan on adding ASP functionality until later.

To publish an .asp file on the Web, save the new file in a virtual directory on your Web site (be sure that the directory has Script or Execute permission enabled). Next, request the file with your browser by typing in the file's URL. (Remember, ASP pages must be served, so you cannot request an .asp file by typing in its physical path.) After the file loads in your browser, you will notice that the server has returned an HTML page. This may seem strange at first, but remember that the server parses and executes all ASP server-side scripts prior to sending the file. The user will always receive standard HTML.

You can use any text editor to create .asp files. As you progress, you may find it more productive to use an editor with enhanced support for ASP, such as Microsoft® Visual InterDev™. (For more information, visit the Microsoft Visual InterDev Web site.)

Adding Server-Side Script Commands

A server-side script is a series of instructions used to sequentially issue commands to the Web server. (If you have developed Web sites previously, then you are probably familiar with client-side scripts, which run on the Web browser.) In .asp files, scripts are differentiated from text and HTML by delimiters. A *delimiter* is a character or sequence of characters that marks the beginning or end of a unit. In the case of HTML, these delimiters are the less than (<) and greater than (>) symbols, which enclose HTML tags.

ASP uses the delimiters <% and %> to enclose script commands. Within the delimiters, you can include any command that is valid for the scripting language you are using. The following example shows a simple HTML page that contains a script command:

```
<HTML>
  <BODY>
  This page was last refreshed on <%= Now() %>.
  </BODY>
</HTML>
```

The VBScript function **Now()** returns the current date and time. When the Web server processes this page, it replaces `<%= Now() %>` with the current date and time and returns the page to the browser with the following result:

```
This page was last refreshed on 01/29/99 2:20:00 PM.
```

Commands enclosed by delimiters are called *primary script commands*, which are processed using the primary scripting language. Any command that you use within script delimiters must be valid for the primary scripting language. By default, the primary scripting language is VBScript, but you can also set a different default language. See Working with Scripting Languages.

If you are already familiar with client-side scripting, you are aware that the HTML <SCRIPT> tag is used to enclose script commands and expressions. You can also use the <SCRIPT> tag for server-side scripting, whenever you need to define procedures in multiple languages within an .asp file. For more information, see Working with Scripting Languages.

Mixing HTML and Script Commands

You can include, within ASP delimiters, any statement, expression, procedure, or operator that is valid for your primary scripting language. A *statement*, in VBScript and other scripting languages, is a syntactically complete unit that expresses one kind of action, declaration, or definition. The conditional **If...Then...Else** statement that appears below is a common VBScript statement:

```
<%
  Dim dtmHour
  dtmHour = Hour(Now())
  If dtmHour < 12 Then
```

```
    strGreeting = "Good Morning!"
  Else
    strGreeting = "Hello!"
  End If
%>
<%= strGreeting %>
```

Depending on the hour, this script assigns either the value "`Good Morning!`" or the value "`Hello!`" to the string variable `strGreeting`. The `<%= strGreeting %>` statement sends the current value of the variable to the browser.

Thus, a user viewing this script before 12:00 noon (in the Web server's time zone) would see this line of text:

```
Good Morning!
```

A user viewing the script at or after 12:00 noon would see this line of text:

```
Hello!
```

You can include HTML text between the sections of a statement. For example, the following script, which mixes HTML within an **If...Then...Else** statement, produces the same result as the script in the previous example:

```
<%
  Dim dtmHour
  dtmHour = Hour(Now())
  If dtmHour < 12 Then
%>
  Good Morning!
<% Else %>
  Hello!
<% End If %>
```

If the condition is true—that is, if the time is before noon—then the Web server sends the HTML that follows the condition ("Good Morning") to the browser; otherwise, it sends the HTML that follows **Else** ("Hello!") to the browser. This way of mixing HTML and script commands is convenient for wrapping the **If...Then...Else** statement around several lines of HTML text. The previous example is more useful if you want to display a greeting in several places on your Web page. You can set the value of the variable once and then display it repeatedly.

Rather than interspersing HTML text with script commands, you can return HTML text to the browser from within a script command. To return text to the browser, use the ASP built-in object **Response**. The following example produces the same result as the previous scripts:

```
<%
  Dim dtmHour
  dtmHour = Hour(Now())
  If dtmHour < 12 Then
    Response.Write "Good Morning!"
  Else
    Response.Write "Hello!"
  End If
%>
```

Response.Write sends the text that follows it to the browser. Use **Response.Write** from within a statement when you want to dynamically construct the text returned to the browser. For example, you might want to build a string that contains the values of several variables. You will learn more about the **Response** object, and objects in general, in Using Components and Objects and Sending Content to the Browser. For now, simply note that you have several ways to insert script commands into an HTML page.

You can include procedures written in your default primary scripting language within ASP delimiters. Refer to Working with Scripting Languages for more information.

If you are working with JScript commands, you can insert the curly braces, which indicate a block of statements, directly into your ASP commands, even if they are interspersed with HTML tags and text. For example:

```
<%
  if (screenresolution == "low")
  {
%>
This is the text version of a page.
<%
  }
  else
  {
%>
This is the multimedia version of a page.
<%
  }
%>
--Or--
<%
  if (screenresolution == "low")
  {
    Response.Write("This is the text version of a page.")
  }
  else
  {
    Response.Write("This is the multimedia version of a page.")
  }
%>
```

Using ASP Directives

ASP provides directives that are not part of the scripting language you use: the output directive and the processing directive.

The ASP *output directive* `<%= expression %>` displays the value of an expression. This output directive is equivalent to using **Response.Write** to display information. For example, the output expression `<%= city %>` displays the word `Baltimore` (the current value of the variable) on the browser.

The ASP *processing directive* `<%@ keyword %>` gives ASP the information it needs to process an .asp file. For example, the following directive sets VBScript as the primary scripting language for the page:

```
<%@ LANGUAGE=VBScript %>
```

The processing directive must appear on the first line of an .asp file. To add more than one directive to a page, the directive must be within the same delimiter. Do not put the processing directive in a file included with the **#include** statement. (For more information, see Including Files.) You must use a space between the at sign (@) and the keyword. The processing directive has the following keywords:

- The LANGUAGE keyword sets the scripting language for the .asp file. See Working with Scripting Languages.
- The ENABLESESSIONSTATE keyword specifies whether an .asp file uses session state. See Managing Sessions.
- The CODEPAGE keyword sets the code page (the character encoding) for the .asp file.
- The LCID keyword sets the locale identifier for the file.
- The TRANSACTION keyword specifies that the .asp file will run under a transaction context. See Understanding Transactions.

Important You can include more than one keyword in a single directive. Keyword/value pairs must be separated by a space. Do not put spaces around the equal sign (=).

The following example sets both the scripting language and the code page:

```
<%@ LANGUAGE="JScript" CODEPAGE="932" %>
```

White Space in Scripts

If your primary scripting language is either VBScript or JScript, ASP removes white space from commands. For all other scripting languages, ASP preserves white space so that languages dependent upon position or indentation are correctly interpreted. White space includes spaces, tabs, returns, and line feeds.

For VBScript and JScript, you can use white space after the opening delimiter and before the closing delimiter to make commands easier to read. All of the following statements are valid:

```
<% Color = "Green" %>

<%Color="Green"%>

<%
Color = "Green"
%>
```

ASP removes white space between the closing delimiter of a statement and the opening delimiter of the following statement. However, it is good practice to use spaces to improve readability. If you need to preserve the white space between two statements, such as when you are displaying the values of variables in a sentence, use an HTML nonbreaking space character (&nbsp;). For example:

```
<%
  'Define two variables with string values.
  strFirstName = "Jeff"
  strLastName = "Smith"
%>
<P>This Web page is customized for "<%= strFirstName %>&nbsp;<%= strLastName %>." </P>
```

Working with Scripting Languages

Programming languages such as Visual Basic, C++, and Java provide low-level access to computer resources and are used to create complex, large-scale programs. Scripting languages, however, are used to create programs of limited capability, called *scripts*, that execute Web site functions on a Web server or browser. Unlike more complex programming languages, scripting languages are *interpreted*, instruction statements are sequentially executed by an intermediate program called a command interpreter. While interpretation reduces execution efficiency, scripting languages are easy to learn and provide powerful functionality. Scripts can be embedded in HTML pages to format content or used to implement COM components encapsulating advanced business logic.

Active Server Pages makes it possible for Web developers to write scripts that execute on the server in variety of scripting languages. In fact, several scripting languages can be used within a single .asp file. In addition, because scripts are read and processed on the server-side, the browser that requests the .asp file does not need to support scripting.

You can use any scripting language for which the appropriate scripting engine is installed on your Web server. A *scripting engine* is a program that processes commands written in a particular language. Active Server Pages comes with two scripting engines: Microsoft Visual Basic Scripting Edition (VBScript) and Microsoft JScript. You can install and use engines for other scripting languages, such as REXX, PERL, and Python.

If you are already a Visual Basic programmer, you can immediately begin using VBScript, which is a subset of Visual Basic. If you are a Java, C, or C++ programmer, you may find that JScript syntax is familiar to you, even though JScript is not directly related to Java or C.

If you are familiar with another scripting language, such as REXX, Perl, or Python you can obtain and install the appropriate scripting engine so that you can use the language you already know. Active Server Pages is a COM scripting host; to use a language you must install a scripting engine that follows the COM Scripting standard and resides as a COM (Component Object Model) object on the Web server.

Setting the Primary Scripting Language

The ASP *primary scripting language* is the language used to process commands inside the <% and %> delimiters. By default, the primary scripting language is VBScript. You can use any scripting language for which you have a script engine as the primary scripting language. You can set the primary scripting language on a page-by-page basis, or for all pages in an ASP application.

Setting the Language for an Application

To set the primary scripting language for all pages in an application, set the **Default ASP Language** property on the **App Options** tab in the Internet Information Services snap-in. For more information, see Configuring ASP Applications.

Setting the Language for a Page

To set the primary scripting language for a single page, add the `<%@ LANGUAGE %>` directive to the beginning of your .asp file. The syntax for this directive is:

```
<%@ LANGUAGE=ScriptingLanguage %>
```

where *`ScriptingLanguage`* is the primary scripting language that you want to set for that particular page. The setting for a page overrides the global setting for all pages in the application.

Follow the guidelines for using an ASP directive; for more information, see Creating an ASP Page.

Note To use a language that does not support the **Object.Method** syntax as the primary scripting language, you must first create the **LanguageEngines** registry key. For more information, see About the Registry.

Using VBScript and JScript on the Server

When using VBScript on the server with ASP, two VBScript features are disabled. Because scripts written with Active Server Pages are executed on the server, the VBScript statements that present user-interface elements, **InputBox** and **MsgBox**, are not supported. In addition, do not use the VBScript functions **CreateObject** and **GetObject** in server-side scripts. Use **Server.CreateObject** instead so that ASP can track the object instance. Objects created by **CreateObject** or **GetObject** cannot access the ASP built-in objects and cannot participate in transactions. The exception to this rule is when you are using the IIS Admin Objects, and when you are using Java monikers. For more information, see Using IIS Admin Objects and Creating an Object from a Java Class.

For a list and description of all VBScript and JScript operators, functions, statements, objects, properties, and methods, refer to the VBScript Language Reference and JScript Language Reference. You can find this reference at the **Windows Script Technologies** Web site.

Including Comments

Because the processing of all scripts in ASP is done on the server side, there is no need to include HTML comment tags to hide the scripts from browsers that do not support scripting, as is often done with client-side scripts. All ASP commands are processed before content is sent to the browser. You can use HTML comments to add remarks to an HTML page; comments are returned to the browser and are visible if the user views the source HTML.

VBScript Comments

Apostrophe-style comments are supported in VBScript. Unlike HTML comments, these are removed when the script is processed and are not sent to the browser.

```
<%
  'This line and the following two are comments.
  'The PrintTable function prints all
  'the elements in an array.
  PrintTable MyArray()
%>
```

You cannot include a comment in an output expression. For example, the first line that follows will work, but the second line will not, because it begins with <%=.

```
<% i = i +1 'This statement increments i. (This script will work.) %>
```

```
<%= name 'This statement prints the variable name. (This script will fail.) %>
```

JScript Comments

The // comment characters are supported in JScript. These characters should be used on each comment line.

```
<%
  var x
  x = new Date()
  // This line sends the current date to the browser,
  // translated to a string.
  Response.Write(x.toString())
%>
```

Case Sensitivity

VBScript is not case sensitive. For example, you can use either **Request** or **request** to refer to the ASP **Request** object. One consequence of case-insensitivity is that you cannot distinguish variable names by case. For example, you cannot create two separate variables named Color and color.

JScript *is* case sensitive. When you use JScript keywords in scripts, you must type the keyword exactly as shown in the reference page for that keyword. For example, using **date** instead of **Date** will cause an error. The case shown in this documentation for the ASP built-in objects will work in JScript commands.

Using Variables and Constants

A *variable* is a named storage location in the computer's memory that contains data, such as a number or a text string. The data contained in a variable is called the variable's *value*. Variables give you a way to store, retrieve, and manipulate values using names that help you understand what the script does.

Declaring and Naming Variables

Follow the rules and guidelines of your scripting language for naming and declaring variables. Even if you are not required to declare a variable before using it, it is a good habit to develop because it helps prevent errors. *Declaring* a variable means telling the script engine that a variable with a particular name exists so that you can use references to the variable throughout a script.

VBScript

VBScript does not require variable declarations, but it is good scripting practice to declare all variables before using them. To declare a variable in VBScript, use the **Dim**, **Public**, or **Private** statement. For example:

```
<% Dim UserName %>
```

You can use the VBScript **Option Explicit** statement in an .asp file to require variables to be explicitly declared with the **Dim**, **Private**, **Public**, and **ReDim** statements. The **Option Explicit** statement must appear after any ASP directives and before any HTML text or script commands. This statement only affects ASP commands that are written in VBScript; it has no effect on JScript commands.

```
<% Option Explicit %>
<HTML>
<%
  Dim strUserName
  Public lngAccountNumber
%>
.
.
.
```

For more information on these commands, see the VBScript Language Reference, which can be found at the **Windows Script Technologies** Web site.

JScript

Although JScript does not usually require variable declarations, it is good scripting practice to declare all variables before using them. To declare a variable, use the **var** statement. For example:

```
<% var UserName %>
```

Typically, you will only need to declare a variable in JScript when you need to distinguish a variable inside a function from a *global* variable used outside the function. In this case, if you do not distinguish between the two variables, JScript will assume that that you referring exclusively to the global variable. For more information on the **var** statement, see the JScript Language Reference. You can find this reference at the **Windows Script Technologies** Web site.

Variable Scope

The *scope*, or lifetime, of a variable determines which script commands can access a variable. A variable declared inside a procedure has *local scope*; the variable is created and destroyed every time the procedure is executed. It cannot be accessed by anything outside the procedure. A variable declared outside a procedure has *global scope*; its value is accessible and modifiable by any script command on an ASP page.

Note Limiting variable scope to a procedure will enhance performance.

If you declare variables, a local variable and a global variable can have the same name. Modifying the value of one will not modify the value of the other. If you do not declare variables, however, you might inadvertently modify the value of a global variable. For example, the following script commands return the value 1 even though there are two variables named Y:

```
<%
  Option Explicit
  Dim Y

  Y = 1
  SetLocalVariable

  Response.Write Y

Sub SetLocalVariable
    Dim Y
    Y = 2
End Sub
%>
```

The following script commands, on the other hand, return the value 2 because the variables are not explicitly declared. When the procedure call sets Y to 2, the scripting engine assumes the procedure intended to modify the global variable:

```
<%
  Option Explicit
  Dim Y = 1

  SetLocalVariable

  Response.Write Y

Sub SetLocalVariable
    Y = 2
End Sub
%>
```

To avoid problems, develop the habit of explicitly declaring all variables. This is particularly important if you use the **#include** statement to include files into your .asp file. The included script is contained in a separate file but is treated as though it were part of the including file. It is easy to forget that you must use different names for variables used in the main script and in the included script unless you declare the variables.

Giving Variables Session or Application Scope

Global variables are accessible only in a single .asp file. To make a variable accessible beyond a single page, give the variable either session or application scope. Session-scoped variables are available to all pages in one ASP application that are requested by one user. Application-scoped variables are available to all pages in one ASP application that are requested by any user. Session variables are a good way to store information for a single user, such as preferences or the user's name or identification. Application variables are a good way to store information for all users of a particular application, such as an application-specific greeting or general values needed by the application.

ASP provides two built-in objects into which you can store variables: the **Session** object and the **Application** object.

You can also create object instances with session or application scope. For more information, see Setting Object Scope.

Session Scope

To give a variable session scope, store it in the **Session** object by assigning a value to a named entry in the object. For example, the following commands store two new variables in the **Session** object:

```
<%
  Session("FirstName") = "Jeff"
  Session("LastName") = "Smith"
%>
```

To retrieve information from the **Session** object, access the named entry by using the output directive (<%=) or **Response.Write**. The following example uses the output directive to display the current value of Session("FirstName"):

```
Welcome <%= Session("FirstName") %>
```

You can store user preferences in the **Session** object, and then access those preferences to determine what page to return to the user. For example, you can allow a user to specify a text-only version of your content in the first page of the application and apply this choice on all subsequent pages that the user visits in this application.

```
<%
  strScreenResolution = Session("ScreenResolution")
  If strScreenResolution = "Low" Then
%>
  This is the text version of the page.
<% Else %>
  This is the multimedia version of the page.
<% End If %>
```

Note   If you refer to a session-scoped variable more than once in a script, consider assigning it to a local variable, as in the previous example, to improve performance.

Application Scope

To give a variable application scope, store it in the **Application** object by assigning a value to a named entry in the object. For example, the following command stores an application-specific greeting in the **Application** object:

```
<% Application("Greeting") = "Welcome to the Sales Department!" %>
```

To retrieve information from the **Application** object, use the ASP output directive (<%=) or **Response.Write** to access the named entry from any subsequent page in the application. The following example uses the output directive to display the value of Application("Greeting"):

```
<%= Application("Greeting") %>
```

Again, if your script repeatedly refers to an application-scoped variable, you should assign it to a local variable to improve performance.

Using Constants

A *constant* is a name that takes the place of a number or string. Some of the base components provided with ASP, such as ActiveX Data Objects (ADO), define constants that you can use in your scripts. A component can declare constants in a *component type library*, a file that contains information about objects and types supported by an COM component. Once you have declared a type library in your .asp file you can use the defined constants in any scripts in the same .asp file. Likewise, you can declare a type library in your Global.asa file to use the defined constants in any .asp file in an application.

To declare a type library, use the <METADATA> tag in your .asp file or Global.asa file. For example, to declare the ADO type library, use the following statements:

```
<!--METADATA NAME="Microsoft ActiveX Data Objects 2.5 Library" TYPE="TypeLib"
UUID="{00000205-0000-0010-8000-00AA006D2EA4}"-->
```

Or, rather than referring to the type library's universal unique indentifier (UUID), you can also refer to the type library by file path:

```
<!-- METADATA TYPE="typelib" FILE="c:\program files\common files\system\ado\msado15.dll"-
->
```

You can then use ADO constants in the .asp file where you declared the type library, or in an .asp file residing to an application containing a Global.asa file with the ADO type library declaration. In the following example, `adOpenKeyset` and `adLockOptimistic` are ADO constants:

```
<%
  'Create and Open Recordset Object.
  Set rstCustomerList = Server.CreateObject("ADODB.Recordset")

  rstCustomerList.ActiveConnection = cnnPubs
  rstCustomerList.CursorType = adOpenKeyset
  rstCustomerList.LockType = adLockOptimistic
%>
```

The following table lists commonly used type libraries and UUIDs:

Type Library	**UUID**
Microsoft ActiveX Data Objects 2.5 Library	{00000205-0000-0010-8000-00AA006D2EA4}

Microsoft CDO 1.2 Library for Windows 2000 Server	{0E064ADD-9D99-11D0-ABE5-00AA0064D470}
MSWC Advertisement Rotator Object Library	{090ACFA1-1580-11D1-8AC0-00C0F00910F9}
MSWC IIS Log Object Library	{B758F2F9-A3D6-11D1-8B9C-080009DCC2FA}

For reference information about the <METADATA> tag, see TypeLibrary Declarations.

In previous versions of ASP, some components provided constant definitions in files that had to be included in each ASP file that used those constants. The use of the **#include** directive to include constant definitions is still supported, but type libraries are generally easier to use and make your scripts more easily upgraded. Components might not provide constant definition files in future releases of ASP.

Note Using the <METADATA> tag rather than the **#include** directive may improve the performance of your Web application.

You can define your own constants. In VBScript, use the **Const** statement. In JScript, you can the **var** statement to assign a constant value to variable. If you want to use your constants on more than one .asp file, put the definitions in a separate file and include them in each .asp file that uses the constants.

Interacting with Client-Side Scripts

ASP's effectiveness can be extended by using it to generate or manipulate client-side scripts. For example, you can write server-side scripts that assemble client-side scripts based on server-specific variables, a user's browser type, or HTTP request parameters.

By interspersing server-side script statements within client-side scripts (enclosed by HTML <SCRIPT> tags), as shown in the following example template, you can dynamically initialize and alter client-side scripts at the request time:

```
<SCRIPT LANGUAGE="VBScript">
<!--

variable = <%=server defined value %>
.
.
.

client-side script

<% server-side script used to generate a client-side statement %>

client-side script
.
.
.
-->
</SCRIPT>
```

Incorporating such functionality can produce some useful and interesting applications. For example, the following is a simple server-side script (written in VBScript) that manipulates a client-side script (written in JScript):

```
<%
  Dim dtmTime, strServerName, strServerSoftware, intGreeting

  dtmTime = Time()
  strServerName = Request.ServerVariables("SERVER_NAME")
  strServerSoftware = Request.ServerVariables("SERVER_SOFTWARE")

  'Generate a random number.
  Randomize
  intGreeting = int(rnd * 3)
%>

  <SCRIPT LANGUAGE="JScript">
  <!--

  //Call function to display greeting
  showIntroMsg()

  function showIntroMsg()
  {
    switch(<%= intGreeting %>)
    {
    case 0:
      msg =  "This is the <%= strServerName%> Web server running <%= strServerSoftware
%>."
      break
    case 1:
      msg = "Welcome to the <%= strServerName%> Web server. The local time is <%= dtmTime
%>."
      break
    case 2:
      msg = "This server is running <%= strServerSoftware %>."
      break
    }

  document.write(msg)

  }

-->
</SCRIPT>
```

Scripts of this kind can be expanded, for example, to configure a client-side database or a DHTML personalization script. Innovative use of this technique can also reduce round-trips and server processing.

Writing Procedures

A *procedure* is a group of script commands that performs a specific task and can return a value. You can define your own procedures and call them repeatedly in your scripts.

You can place procedure definitions in the same .asp file that calls the procedures, or you can put commonly used procedures in a shared .asp file and use the **#include** directive to include it in other .asp files that call the procedures. Alternatively, you could encapsulate the functionality in a COM component.

Defining Procedures

Procedure definitions can appear within <SCRIPT> and </SCRIPT> tags and must follow the rules for the declared scripting language. Use the <SCRIPT> element for procedures in languages other than the primary scripting language. However, use the scripting delimiters (<% and %>) for procedures in the primary scripting language.

When you use the HTML <SCRIPT> tag, you must use two attributes to ensure server-side processing of the script. The syntax for using the <SCRIPT> tag is:

```
<SCRIPT LANGUAGE=JScript RUNAT=SERVER>

  procedure definition

</SCRIPT>
```

The RUNAT=SERVER attribute tells the Web server to process the script on the server. If you do not set this attribute, the script is processed by the client browser. The LANGUAGE attribute determines the scripting language used for this script block. You can specify any language for which you have the scripting engine installed with the server. To specify VBScript, use the value VBScript. To specify JScript, use the value JScript. If you do not set the LANGUAGE attribute, the script block is interpreted in the primary scripting language.

The commands in the script block must form one or more complete procedures in the chosen scripting language. For example, the following commands define the JScript procedure **MyFunction**.

```
<HTML>
<SCRIPT LANGUAGE=JScript RUNAT=SERVER >
  function MyFunction()
  {
    Response.Write("You called MyFunction().")
  }
</SCRIPT>
```

Important Do not include within server-side <SCRIPT> tags any script commands that are not part of complete procedures. Commands that are not part of a procedure may cause unpredictable results because the code is executed in the following order: Script blocks in non-default languages are executed in order of appearance, then the inline code, and finally the script blocks in the default language. In addition, you cannot use the ASP output directive <%= %> within a procedure. Instead, use **Response.Write** to send content to the browser.

Calling Procedures

To call procedures, include the name of the procedure in a command. If you are calling JScript procedures from VBScript, you must use parentheses after the procedure name; if the procedure has no arguments, use empty parentheses. If you are calling either VBScript or JScript procedures from JScript, always use parentheses after the procedure name.

For VBScript, you can also use the **Call** keyword when calling a procedure. However, if the procedure that you are calling requires arguments, the argument list must be enclosed in parentheses. If you omit the **Call** keyword, you also must omit the parentheses around the argument list. If you use **Call** syntax to call any built-in or user-defined function, the function’s return value is discarded.

The following example illustrates creating and calling procedures by using two different scripting languages (VBScript and JScript).

```
<%@ LANGUAGE=VBScript %>
<HTML>
  <BODY>
  <!-- Call the JScript procedure from within VBScript-->
  <% call printDate() %>
  <!--Call the VBScript procedure from within VBScrip-->
<% Echo %>
  <BR>
  </BODY>
</HTML>

<%Sub Echo%>
<!--Note: this will not output anything unless the page is called with a query string
like http://localhost/test.asp?x=1%20have&y=a%20cunning&z=plan -->
<%
  Response.Write "<TABLE BORDER=1>" & _
    "<TR><TH>Name</TH><TH>Value</TH></TR>"

  Set objQueryString = Request.QueryString

  For Each strSelection In objQueryString
    Response.Write "<TR><TD>" & strSelection & "</TD><TD>" & _
    objQueryString(strSelection) & "</TD></TR>"
  Next

  Response.Write "</TABLE>"

End Sub
%>

<SCRIPT LANGUAGE=JScript RUNAT=SERVER>

function printDate()
{
  var x

  x = new Date()

  Response.Write(x.toString())
  Response.Write("<BR>")
}
</SCRIPT>
```

Note VBScript calls to JScript functions are *not* case sensitive.

Passing Arrays to Procedures

To pass an entire array to a procedure in VBScript, use the array name followed by empty parentheses; in JScript, use empty square brackets.

Working with Collections

Most of the ASP built-in objects provide collections. *Collections* are data structures similar to arrays that store strings, numbers, objects and other values. Unlike arrays, collections expand and contract automatically as items are retrieved or stored. The position of an item will also move as the collection is modified. You can access an item in a collection by its unique string key, by its index (position) in the collection, or by iterating through all the items in the collection.

Accessing an Item by Name or Index

You can access a specific item in a collection by referencing its unique string key, or name. For example, the **Contents** collection holds any variables stored in the **Session** object. It can also hold any objects instantiated by calling **Server.CreateObject**. Suppose you have stored the following user information in the **Session** object:

```
<%
  Session.Contents("FirstName") = "Meng"
  Session.Contents("LastName") = "Phua"
  Session.Contents("Age") = 29
%>
```

You can access an item by using the string key you associated with the item when you stored it in the collection. For example, the following expression returns the string "Meng":

```
<%= Session.Contents("FirstName") %>
```

You could also access an item by using the index, or number, associated with an item. For example, the following expression retrieves the information stored in the second position of the **Session** object and returns "Phua":

```
<%= Session.Contents(2) %>
```

ASP collections are numbered starting with 1. The index associated with an item might change as items are added to or removed from the collection. You should not depend on an item’s index remaining the same. Indexed access is generally used to iterate through a collection, as described in the following topics, or to access an item in a read-only collection.

You can also use a shorthand notation to access items by name. ASP searches the collections associated with an object in a particular order. If an item with a particular name appears only once in an object's collections, you can eliminate the collection name (although doing so may affect performance):

```
<%= Session("FirstName") %>
```

Eliminating the collection name is generally safe when you are accessing items stored in the **Application** or **Session** object. For the **Request** object, however, it is safer to specify the collection name because the collections could easily contain items with duplicate names.

Iterating through a Collection

You can iterate through all the items in a collection to see what is stored in the collection or to modify the items. You must supply the collection name when you iterate through a collection. For example, you can use the VBScript **For...Each** statement to access the items you stored in the **Session** object:

```
<%
  'Declare a counter variable.
  Dim strItem

  'For each item in the collection, display its value.
  For Each strItem In Session.Contents
    Response.Write Session.Contents(strItem) & "<BR>"
  Next
%>
```

You can also iterate through a collection by using the VBScript **For...Next** statement. For example, to list the three items stored in **Session** by the previous example, use the following statements:

```
<%
  'Declare a counter variable.
  Dim intItem
  'Repeat the loop until the value of counter is equal to 3.
  For intItem = 1 To 3
    Response.Write Session.Contents(intItem) & "<BR>"
  Next
%>
```

Because you do not usually know how many items are stored in a collection, ASP supports the **Count** property for a collection, which returns the number of items in the collection. You use the **Count** property to specify the end value of the counter.

```
<%
  'Declare a counter variable.
  Dim lngItem, lngCount

  lngCount = Session.Contents.Count

  'Repeat this loop until the counter equals the number of items
  'in the collection.
  For lngItem = 1 To lngCount
     Response.Write Session.Contents(lngItem) & "<BR>"
  Next
%>
```

In JScript, you use the **for** statement to loop through a collection. For greater efficiency when using the **Count** property with a JScript **for** statement, you should assign the value of **Count** to a local variable and use that variable to set the end value of the counter. That way, the script engine does not have to look up the value of **Count** each time through the loop. The following example demonstrates this technique:

```
<%
  var intItem, intNumItems;

  intNumItems = Session.Contents.Count;

  for (intItem = 1; intItem <= intNumItems; intItem++)
  {
    Response.Write(Session.Contents(intItem) + "<BR>")
  }
%>
```

Microsoft JScript supports an **Enumerator** object that you can also use to iterate through an ASP collection. The **atEnd** method indicates whether there are any more items in the collection. The **moveNext** method moves to the next item in the collection.

```
<%
  Session.Contents("Name") = "Suki White"
  Session.Contents("Department") = "Hardware"
                  .
                  .
                  .
  //Create an Enumerator object.
  var mycollection = new Enumerator(Session.Contents);

  //Iterate through the collection and display each item.
  while (!mycollection.atEnd())
  {
    var x  = myCollection.item();
    Response.Write(Session.Contents(x) + "<BR>");
    myCollection.moveNext();
  }
%>
```

Iterating through a Collection with Subkeys

Scripts might embed several related values in a single cookie to reduce the number of cookies passed between the browser and the Web server. The **Cookies** collection of the **Request** and **Response** objects can thus hold multiple values in a single item. These subitems, or subkeys, can be accessed individually. Subkeys are supported only by the **Request.Cookies** and **Response.Cookies** collections. **Request.Cookies** supports only read operations; **Response.Cookies** supports only write operations.

The following creates a regular cookie and a cookie with a subkeys:

```
<%
  'Send a cookie to the browser.
  Response.Cookies("Fruit") = "Pineapple"

  'Send a cookie with subkeys to browser.
  Response.Cookies("Mammals")("Elephant") = "African"
  Response.Cookies("Mammals")("Dolphin") = "Bottlenosed"
%>
```

The cookie text in the HTTP response sent to the browser would appear as the following:

```
HTTP_COOKIE= Mammals=ELEPHANT=African&DOLPHIN=Bottlenosed; Fruit=Pineapple;
```

You can also enumerate all the cookies in the **Request.Cookies** collection and all the subkeys in a cookie. However, iterating through subkeys on a cookie that does not have subkeys will not produce an item. You can avoid this situation by first checking to see whether a cookie has subkeys by using the **HasKeys** attribute of the **Cookies** collection. This technique is demonstrated in the following example.

```
<%
   'Declare counter variables.
   Dim Cookie, Subkey

   'Display the entire cookie collection.
   For Each Cookie In Request.Cookies
     Response.Write Cookie
     If Request.Cookies(Cookie).HasKeys Then
       Response.Write "<BR>"
       'Display the subkeys.
       For Each Subkey In Request.Cookies(Cookie)
         Response.Write " ->" & Subkey & " = " & Request.Cookies(Cookie)(Subkey) & "<BR>"
       Next
     Else
       Response.Write " = " & Request.Cookies(Cookie) & "<BR>"
     End If
   Next
%>
```

This script would return the following results:

```
Mammals

->ELEPHANT = African

->DOLPHIN = Bottlenosed

Fruit = Pineapple
```

The Case of the Key Name

The **Cookies**, **Request**, **Response**, and **ClientCertificate** collections can reference the same, unique string key name. For example, the following statements reference the same key name, *User*, but return different values for each collection:

```
strUserID = Request.Cookies("User")

strUserName = Request.QueryString("User")
```

The case of the key name is set by the first collection to assign a value to the key. Examine the following script:

```
<%
  'Retrieve a value from QueryString collection using the UserID key.
  strUser = Request.QueryString("UserID")

  'Send a cookie to the browser, but reference the key, UserId, which has a different
spelling.
  Response.Cookies("UserId")= "1111-2222"
  Response.Cookies("UserId").Expires="December 31, 2000"
%>
```

Although it may appear that key name *UserId* was assigned to the cookie, in actuality, the key name *UserID* (which is capitalized differently) was assigned to the cookie. The **QueryString** collection was first to define the case of this key.

Because references to collection values are independent of the case of the key name, this behavior will not affect your scripts unless your application uses case sensitive processing of key names.

Iterating through a Collection of Objects

The **Session** and **Application** collections can hold either scalar variables or object instances. The **Contents** collection holds both scalar variables and object instances created by calling **Server.CreateObject**. The **StaticObjects** collection holds objects created by using the HTML <OBJECT> tag within the scope of the **Session** object. For more information about instantiating objects in this manner, see Setting Object Scope.

When you iterate through a collection that contains objects, you can either access the object's Session or Application state information or access the object's methods or properties. For example, suppose your application uses several objects to create a user account, and each object has an initialization method. You could iterate through the **StaticObjects** collection to retrieve object properties:

```
<%
  For Each Object in Session.StaticObjects
    Response.Write Object & ": " & Session.StaticObjects(Object)
  Next
%>
```

What's Different About ASP Collections?

Although the ASP collections described in this topic are similar to the Visual Basic **Collection** object, there are some differences. The ASP collections support the **Count** property and the **Item**, **Remove**, and **RemoveAll** methods. They do not support the **Add** method.

Processing User Input

By using the ASP **Request** object, you can create simple, yet powerful scripts for collecting and processing data gathered with HTML forms. In this topic, you will not only learn how to create basic form processing scripts, but also acquire useful techniques for validating form information, both on your Web server and at the user's browser.

About HTML Forms

HTML forms, the most common method for gathering Web-based information, consist of arrangements of special HTML tags that render user interface elements on a Web page. Text boxes, buttons, and check boxes are examples of elements that enable users to interact with a Web page and submit information to a Web server.

For example, the following HTML tags generate a form where a user can enter their first name, last name, and age, and includes a button for submitting information to a Web server. The form also contains an hidden input tag (not displayed by the Web browser) that you can use to pass additional information to a Web server.

```
<FORM METHOD="Get" ACTION="Profile.asp">
<INPUT TYPE="Text" NAME="FirstName">
<INPUT TYPE="Text" NAME="LastName">
<INPUT TYPE="Text" NAME="Age">
<INPUT TYPE="Hidden" NAME="UserStatus" VALUE="New">
<INPUT TYPE="Submit" VALUE="Enter">
</FORM>
```

Detailing the complete set of HTML form tags is outside the scope of this topic, however, there are numerous sources of information that you can use to learn about creating useful and engaging HTML forms. For example, use your Web browser's source viewing feature to examine how HTML forms are created on other Web sites. Also, visit **MSDN Online** to learn innovative ways of using HTML forms with other Internet technologies.

Processing Form Inputs with ASP

After creating an HTML form, you will need to process user input, which means sending the information to an .asp file for parsing and manipulation. Once again, examine the HTML code from the previous example. Notice that the <FORM> tag's ACTION attribute refers to a file called Profile.asp. When the user submits HTML information, the browser uses the POST method to send to the information to an .asp file on the server, in this case Profile.asp. The .asp file may contain scripts that process information and interact with other scripts, COM components, or resources, such as a database.

Using ASP, there are three basic ways to collect information from HTML forms:

- A static .htm file can contain a form that posts its values to an .asp file.
- An .asp file can create a form that posts information to another .asp file.
- An .asp file can create a form that posts information to itself, that is, to the .asp file that contains the form.

The first two methods operate in the same way as forms that interact with other Web server programs, except that with ASP, the task of collecting and processing form information is greatly simplified. The third method is a particularly useful and will be demonstrated in the Validating Form Input section.

Getting Form Input

The ASP **Request** object provides two collections that facilitate the task of retrieving form information sent with as a URL request.

The QueryString Collection

The **QueryString** collection retrieves form values passed to your Web server as text following a question mark in the request URL. The form values can be appended to the request URL by using either the HTTP GET method or by manually adding the form values to the URL.

For example, if the previous form example used the GET method (METHOD="GET") and the user typed *Clair*, *Hector*, and *30*, then the following URL request would be sent to the server:

```
http://Reskit/Workshop1/Painting/Profile.asp?FirstName=Clair&LastName=Hector&Age=30&UserS
tatus=New
```

Profile.asp might contain the following form processing script:

```
Hello <%= Request.QueryString("FirstName") %> <%= Request.QueryString("LastName") %>.
You are <%= Request.QueryString("Age") %> years old!

<%
  If Request.QueryString("UserStatus") = "New" Then
    Response.Write "This is your first visit to this Web site!"
  End if
%>
```

In this case, the Web server would return the following text to the user's Web browser:

```
Hello Clair Hector. You are 30 years old! This is your first visit to this Web site!
```

The **QueryString** collection also has an optional parameter that you can use to access one of multiple values that appear in the URL request (using the GET method). You can also use the **Count** property to count the number of times that a specific type of value appears.

For example, a form containing a list box with multiple items can generate the following request:

```
http://Reskit/OrganicFoods/list.asp?Food=Apples&Food=Olives&Food=Bread
```

You could use the following command to count multiple values:

```
Request.QueryString("Food").Count
```

To display the multiple values types, List.asp could contain the following script:

```
<%
  lngTotal = Request.QueryString("Food").Count
  For i = 1 To lngTotal
    Response.Write Request.QueryString("Food")(i) & "<BR>"
  Next
%>
```

The preceding script would display:

```
Apples
Olives
Bread
```

You can also display the entire list of values as a comma-delimited string by using the following:

```
<% Response.Write Request.QueryString("Item") %>
```

This would display the following string:

```
Apples, Olives, Bread
```

Form Collection

When you use the HTTP GET method to pass long and complex form values to a Web server, you run the risk of losing information. Some Web servers tend to restrict the size of the URL query string, so that lengthy form values passed with the GET method might be truncated. If you need to send a large amount of information from a form to a Web server, you should use the HTTP POST method. The POST method, which sends form data in the HTTP request body, can send a an almost unlimited number of characters to a server. You can use the ASP **Request** object's **Form** collection to retrieve the values sent with the POST method.

The **Form** collection stores values in a manner similar to the **QueryString** collection. For example, if a user filled out a form by entering a long list of names, then you could retrieve the food names with the following script:

```
<%
  lngTotal = Request.Form("Food").Count
  For i = 1 To lngTotal
   Response.Write Request.Form("Food")(i) & "<BR>"
  Next
%>
```

Validating Form Input

A well-designed Web form often includes a client script that validates user input prior to sending information to the server. *Validation scripts* can check for such things as whether the user entered a valid number or whether a text box was left empty. Imagine that your Web site includes a form that enables users to compute the rate of return on an investment. You will probably want to verify whether a user has actually entered numerical or text information in the appropriate form fields, prior to sending potentially invalid information to your server.

In general, it's good practice to do as much form validation as possible on the client side. Beyond prompting users more quickly about input errors, client-side validation yields faster response times, reduces server loads, and frees bandwidth for other applications.

The following client-side script validates user–input (in this case, the script determines whether an account number entered by the user is actually a number) prior to sending information to the server:

```
<SCRIPT LANGUAGE="JScript">

function CheckNumber()
{
 if (isNumeric(document.UserForm.AcctNo.value))
   return true
 else
 {
   alert("Please enter a valid account number.")
   return false
 }
}

//Function for determining if form value is a number.
//Note:  The JScript isNaN method is a more elegant way to determine whether
//a value is not a number. However, some older browsers do not support this method.
function isNumeric(str)
{
  for (var i=0; i < str.length; i++)
                  {
    var ch = str.substring(i, i+1)
    if( ch < "0" || ch>"9" || str.length == null)
                                    {
      return false
    }
  }
  return true
}
</SCRIPT>

<FORM METHOD="Get" ACTION="balance.asp" NAME="UserForm" ONSUBMIT="return CheckNumber()">

         <INPUT TYPE="Text"   NAME="AcctNo">
         <INPUT TYPE="Submit" VALUE="Submit">

</FORM>
```

If form validation requires database access, however, you should consider using server-side form validation. A particularly advantageous way of carrying out server-side validation is to create a form that posts information to itself. That is, the .asp file actually contains the HTML form that retrieves user input. (Remember, you can use ASP to interact with client-side scripts and HTML. For more information, see Interacting with Client-Side Scripts.) The input is returned to the same file, which then validates the information and alerts the user in case of an invalid input.

Using this method of processing and validating user input can greatly enhance the usability and responsiveness of your Web based forms. For example, by placing error information adjacent to the form field where invalid information was entered, you make it easier for the user to discover the source of the error. (Typically, Web-based forms forward requests to a separate Web page containing error information. Users who do not immediately understand this information may become frustrated.)

For example, the following script determines whether a user entered a valid account number by posting information to itself (Verify.asp) and calling a user defined function that queries a database:

```
<%
  strAcct = Request.Form("Account")
  If Not AccountValid(strAcct) Then
    ErrMsg = "<FONT COLOR=Red>Sorry, you may have entered an invalid account
number.</FONT>"
  Else
    Process the user input
    .
    .
    .
    Server.Transfer("Complete.asp")
  End If

  Function AccountValid(strAcct)
```

```
    A database connectivity script or component method call goes here.
  End Function
%>

<FORM METHOD="Post"  ACTION="Verify.asp">
Account Number:  <INPUT TYPE="Text" NAME="Account"> <%= ErrMsg %> <BR>
<INPUT TYPE="Submit">
</FORM>
```

In this example, the script is located in a file named Verify.asp, the same file that contains the HTML form; it posts information to itself by specifying Verify.asp in the ACTION attribute.

Important If your are using JScript for server-side validation, be sure to place a pair of empty parentheses following the **Request** collection item (either **QueryString** or **Form**) when you are assigning the collection to a local variable. Without parenthesis, the collection returns an object, rather than a string. The following script illustrates the correct way to assign variables with JScript:

```
<%
   var Name = Request.Form("Name")();
   var Password = Request.Form("Password")();

  if(Name > "")
  {
     if(Name == Password)
      Response.Write("Your name and password are the same.")
  else
      Response.Write("Your name and password are different.");
  }
%>
```

VBScript exhibits the same behavior if the collection contains multiple values that are comma-separated or indexable. This means that for both VBScript and JScript, in addition to placing a pair of empty parentheses following the **Request** collection item, you will need to specify the index of the desired value. For example, the following line of JScript returns only the first of multiple values for a form element:

```
var Name = Request.Form("Name")(1);
```

Using Components and Objects

COM components are the key to building powerful, real-world Web applications. Components provide functionality that you use in your scripts to perform specialized tasks, such as executing financial transactions or validating data. ASP also provides a set of base components that you can use to greatly enhance your scripts.

About Components

A *COM component* is a reusable, programmatic building block that contains code for performing a task or set of tasks. Components can be combined with other components (even across networks) to create a Web application. COM components execute common tasks so that you do not have to create your own code to perform these tasks. For example, you can use a stock ticker component to display the latest stock quotes on a Web page. However, it would be difficult to create a script that provides the same functionality. Also, the script would not be as reusable as a component.

If you are new to scripting, you can write scripts that use components without knowing anything about how the component works. ASP comes with base components that you can use immediately. For example, you can use the ActiveX Data Objects (ADO) components to add database connectivity to your Web pages. Additional components can also be obtained from third-party developers.

If you are a Web application developer, components are the best way to encapsulate your business logic into reusable, secure modules. For example, you could use a component to verify credit card numbers by calling the component from a script that processes sales orders. Because the verification is isolated from the order process, you can update the component when the credit card verification process changes, without changing your order process. Also, since COM components are reusable, you could reuse the component in other scripts and applications. Once you have installed a component on your Web server, you can call it from a ASP server-side script, an ISAPI extension, another component on the server, or a program written in another COM-compatible language.

You can create components in any programming language that supports the Component Object Model (COM), such as C, C++, Java, Visual Basic, or numerous scripting languages. (If you are familiar with COM programming, COM components are also known as Automation servers.) To run on the Web server, your COM components cannot have any graphical user interface elements, such as the Visual Basic **MsgBox** function; graphical interface elements would only be viewable on the server, and not the browser.

Creating an Instance of a Component's Object

A component is executable code contained in a dynamic-link library (.dll) or in an executable (.exe) file. Components provide one or more *objects*, self contained units of code which perform specific functions within the component. Each object has methods (programmed procedures) and properties (behavioral attributes). To use an object provided by a component, you create an instance of the object and assign the new instance to a variable name. Use the ASP **Server.CreateObject** method or the HTML <OBJECT> tag to create the object instance. Use your scripting language's variable assignment statement to give your object instance a name. When you create the object instance, you must provide its registered name (PROGID). For the base components provided with ASP, you can get the PROGID for the objects from the reference pages (see Installable Components for ASP).

For example, the Ad Rotator component randomly rotates through a series of graphical advertisements. The Ad Rotator component provides one object, called the Ad Rotator object, whose PROGID is "MSWC.AdRotator." To create an instance of the Ad Rotator object, use one of the following statements:

VBScript:

```
<% Set MyAds = Server.CreateObject("MSWC.AdRotator") %>
```

JScript:

```
<% var MyAds = Server.CreateObject("MSWC.AdRotator") %>
```

If you are already familiar with VBScript or JScript, note that you do not use the scripting language's function for creating a new object instance (**CreateObject** in VBScript or **New** in JScript). You must use the ASP **Server.CreateObject** method; otherwise, ASP cannot track your use of the object in your scripts.

You can also use the HTML <OBJECT> tag to create an object instance. You must supply the RUNAT attribute with a value of Server, and you must provide the ID attribute set to the variable name you will use in your scripts. You can identify the object by using either its registered name (PROGID) or its registered number (CLSID).The following example uses the registered name (PROGID) to create an instance of the Ad Rotator object:

```
<OBJECT RUNAT=Server ID=MyAds PROGID="MSWC.AdRotator"></OBJECT>
```

The following example uses the registered number (CLSID) to create an instance of the Ad Rotator object:

```
<OBJECT RUNAT=SERVER ID=MyAds

CLASSID="Clsid:1621F7C0-60AC-11CF-9427-444553540000"></OBJECT>
```

Use Scripting to Create COM Components

ASP supports Windows Script Components, Microsoft's powerful scripting technology that you can use to create COM components. Specifically, you can encapsulate common scripts, such as those used for database access or content generation, into reusable components accessible from any .asp file or program. You can create Windows Script Components by writing scripts in a language such as VBScript or JScript without a special development tool. You can also incorporate Windows Script Components into programs written in COM compliant programming languages, such as Visual Basic, C++, or Java.

The following is an example of a Windows Script Components, written in VBScript, that defines methods for converting temperature measurements between Fahrenheit and Celsius:

```
<SCRIPTLET>

<Registration
         Description="ConvertTemp"
         ProgID="ConvertTemp.Scriptlet"
         Version="1.00"
>
</Registration>

<implements id=Automation type=Automation>
         <method name=Celsius>
                  <PARAMETER name=F/>
         </method>
         <method name=Fahrenheit>
                  <PARAMETER name=C/>
         </method>
</implements>

<SCRIPT LANGUAGE=VBScript>

  Function Celsius(F)
           Celsius = 5/9 * (F - 32)
  End Function

  Function Fahrenheit(C)
           Fahrenheit = (9/5 * C) + 32
  End Function

</SCRIPT>
</SCRIPTLET>
```

Before implementing this Windows Script Component you must save this file with an .sct extension and then in Windows Explorer, right-click this file and select **Register**. To use this Windows Script Component in a Web page, you would use a server-side script such as the following:

```
<%
  Option Explicit

  Dim objConvert
  Dim sngFvalue, sngCvalue

  sngFvalue = 50
  sngCvalue = 21

  Set objConvert = Server.CreateObject("ConvertTemp.Scriptlet")
%>

<%= sngFvalue %> degrees Fahrenheit is equivalent to <%= objConvert.Celsius(sngFvalue) %>
degrees Celsius<BR>

<%= sngCvalue %> degrees Celsius is equivalent to <%= objConvert.Fahrenheit(sngCValue) %>
degrees Fahrenheit<BR>
```

Using ASP Built-in Objects

ASP also provides built-in objects for performing useful tasks that simplify Web development. For example, you can use the **Request** object to easily access information associated with an HTTP request, such as user input coming from HTML forms or cookies. Unlike using the objects provided by a COM component, you do not need to create an instance of an ASP built-in object to use it in your scripts. These objects are automatically created for you when the ASP request starts processing. You access the methods and properties of a built-in object in the same way in which you access the methods and properties of a component's objects, as described in this topic. For a complete description of the built-in objects, see Active Server Pages Objects Quick Reference Card.

Calling an Object Method

A *method* is an action you can perform on an object or with an object. The syntax for calling a method is:

Object.Method *parameters*

The *parameters* vary depending on the method.

For example, you can use the **Write** method of the **Response** built-in object to send information to the browser as shown in the following statement:

```
<% Response.Write "Hello World" %>
```

Note Some scripting languages do not support the **Object.Method** syntax. If your language does not, you must add an entry to the registry in order to use that language as your primary scripting language. See Working with Scripting Languages for more information.

Setting an Object Property

A *property* is an attribute that describes the object. Properties define object characteristics, such as the type of the object, or describe the state of an object, such as enabled or disabled. The syntax is:

Object.Property

You can sometimes read and set the value of a property. In addition, for some objects, you can also add new properties.

For example, the Ad Rotator component has a property, **Border**, which specifies whether the ad has a border around it and determines the border thickness. The following expression specifies no border:

```
<% MyAds.Border = 0 %>
```

For some properties, you can display the current value by using the ASP output directive. For example, the following statement returns TRUE if the browser is still connected to the server:

```
<%= Response.IsClientConnected %>
```

Creating an Object from a Java Class

To use **Server.CreateObject** to create an instance of a Java class, you must use the JavaReg program to register the class as a COM component. You can then use **Server.CreateObject** method or an HTML <OBJECT> tag with the PROGID or CLSID.

Alternatively, you can use the mechanism provided by Java monikers to instantiate the Java class directly without using the JavaReg program. To instantiate a class with monikers, use the VBScript or JScript **GetObject** statement and provide the full name of the Java class in the form `java:`*`classname`*. The following VBScript example creates an instance of the Java **Date** class.

```
<%
  Dim dtmDate
  Set dtmDate = GetObject("java:java.util.Date")
%>
The date is <%= dtmDate.toString() %>
```

Objects created by calling **GetObject** instead of **Server.CreateObject** can also access the ASP built-in objects and participate in a transaction. To use Java monikers, however, you must be using version 2.0, or later, of the Microsoft virtual machine.

Setting Object Scope

The scope of an object determines which scripts can use that object. By default, when you create an object instance, the object has page scope. Any script command in the same ASP page can use a page-scope object; the object is released when the .asp file completes processing the request. The recommended scope for most objects is page scope. You can change the scope of an object, however, to make it accessible from scripts on other pages. This topic explains how to work with page scope objects and how to change the scope of objects.

Using Page Scope Objects

An object that you create by using **Server.CreateObject** or the HTML <OBJECT> tag on an ASP page exists for the duration of that page. The object is accessible to any script commands on that page, and it is released when ASP has finished processing the page. Thus an object has the scope, or lifetime, of a page.

Creating Objects in Loops

In general, you should avoid creating objects inside a loop. If you must create objects in a loop, you should manually release the resources used by an object:

```
<%
  Dim objAd
  For i = 0 To 1000
    Set objAd = Server.CreateObject("MSWC.AdRotator")
    .
    .
    .
    objAd.GetAdvertisement
    .
    .
    .
    Set objAd = Nothing
  Next
%>
```

Giving an Object Session Scope

A *session-scope* object is created for each new session in an application and released when the session ends; thus, there is one object per active session. Session scope is used for objects that are called from multiple scripts but affect one user session. You should give objects session scope only when needed. If you do use session scope, you must know the threading model of the component that provides the object because the threading model affects the performance and security context of the object. For more information, see Advanced Information: Performance Issues in this topic.

To give an object session scope, store the object in the ASP **Session** built-in object. You can use either the HTML <OBJECT> tag in a Global.asa file or the **Server.CreateObject** method on an ASP page to create a session scope object instance.

In the Global.asa file, you can use the <OBJECT> tag, extended with RUNAT attribute (which must be set to SERVER) and the SCOPE attribute (which must be set to Session). The following example creates a session-scope instance of the Browser Type object of the Browser Capabilities component:

```
<OBJECT RUNAT=SERVER SCOPE=Session ID=MyBrowser PROGID="MSWC.BrowserType">


</OBJECT>
```

Once you have stored the object in the **Session** object, you can access the object from any page in the application. The following statement uses the object instance created by the <OBJECT> tag in the previous example:

```
<%= If MyBrowser.browser = "IE"  and  MyBrowser.majorver >= 4  Then . . .%>
```

On an ASP page, you can also use the **Server.CreateObject** method to store an object in the **Session** built-in object. The following example stores an instance of the Browser Type object in the **Session** object.

```
<% Set Session("MyBrowser") = Server.CreateObject("MSWC.BrowserType") %>
```

To display browser information in a different .asp file, you first retrieve the instance of the **BrowserType** object stored in the **Session** object, and then call the **Browser** method to display the name of the browser:

```
<% Set MyBrowser = Session("MyBrowser") %>


<%= MyBrowser.browser %>
```

ASP does not instantiate an object that you declare with the <OBJECT> tag until that object is referenced by a script command from an .asp file. The **Server.CreateObject** method instantiates the object immediately. Thus, the <OBJECT> tag offers better scalability than the **Server.CreateObject** method for session-scope objects.

Giving an Object Application Scope

An *application-scope* object is a single instance of an object that is created when the application starts. This object is shared by all client requests. Some utility objects, such as the objects of the Page Counter Component, might perform better in application scope, but generally you should use the alternatives proposed in the following section. In addition, the threading model affects the performance and security context of the object (see Advanced Information: Performance Issues in this topic).

To give an object application scope, store the object in the ASP **Application** built-in object. You can use either the <OBJECT> tag in a Global.asa file or the **Server.CreateObject** method in an .asp file to create an application scope object instance.

In the Global.asa file, you can use the <OBJECT> tag, extended with RUNAT attribute (which must be set to Server) and the SCOPE attribute (which must be set to Application). For example, the following is an example of using the <OBJECT> tag to create an application-scope instance of the Ad Rotator object:

```
<OBJECT RUNAT=SERVER SCOPE=Application ID=MyAds PROGID="MSWC.AdRotator">


</OBJECT>
```

After storing the Ad Rotator object in Application state, you can access from any page in you application using a statement such as the following:

```
<%=MyAds.GetAdvertisement("CustomerAds.txt") %>
```

Alternatively, in an .asp file, you can use **Server.CreateObject** to store an object instance in the **Application** built-in object, such as in the following example:

<% Set Application("MyAds") = Server.CreateObject("MSWC.Adrotator")%>

You can display the advertisement in your application's .asp files by retrieving the instance of the Ad Rotator object from Application state, as in the following example:

<%Set MyAds = Application("MyAds") %> <%=MyAds.GetAdvertisement("CustomerAds.txt") %>

Alternatives to Session and Application Scope

In general, you should try to extensively use application or session state for items or objects that take a long time to initialize, such as dictionary objects or recordsets. However, if you find that objects in session or application state are consuming too many resources, such as memory or database connections, you should seek alternative ways to implement these objects. For example, the threading model of a component can affect the performance of objects you create from it, especially objects with session or application scope.

In many cases, a better solution than creating application or session scope objects is to use session or application scope variables that pass information to objects created at the page level. For example, you should not give an ADO **Connection** object session or application scope because the connection it creates remains open for a long time and because your script no longer takes advantage of connection pooling. You can, however, store an ODBC or OLE DB connection string in the **Session** or **Application** built-in object and access the string to set a property on the **Connection** object instance that you create on a page. In this way, you store frequently used information in session or application state but you create the object that uses the information only when needed. For more information about scoping variables, see Using Variables and Constants.

User-Defined JScript Objects

You can create your own JScript object by defining a constructor function that creates and initializes the properties and methods of the new object. The object instance is created when your script uses the **new** operator to invoke the constructor. User-defined JScript objects are supported in ASP server-side scripts and work well when they have page scope. However, you cannot give a user-defined object application scope. Also, giving a user-defined JScript object session scope may affect the functionality of the object. In particular, if an object has session scope, scripts on other pages can access the object's properties but cannot call its methods. Also, giving a user-defined JScript object session scope can affect Web application performance.

Advanced Information: Performance Issues

The threading model of a component may affect the performance of your Web site. Generally, objects that are marked Both are the recommended objects to use in .asp files if they will be stored in **Session** and **Application** state. Single-threaded, Apartment, and free-threaded objects are not recommended.

Because you may not always have control over the threading model of the objects you use, the following guidelines will help you get the best performance:

- **Page scope objects** Objects marked Both or Apartment will give you the best performance.
- **Application scope objects** Objects marked Both, that also aggregate the FreeThreadedMarshaler, will give you the best performance. You can use either the <OBJECT> tag or the **Server.CreateObject** method to store objects marked Both in the **Application** object. You must use the HTML <OBJECT> tag with apartment-threaded objects.
- **Session scope objects** Objects marked Both will give you the best performance. Using single-threaded or apartment-threaded objects will cause the Web server to lock the session down to one thread. Free-threaded objects do not lock down the session, but are slow. You can use either the <OBJECT> tag or the **Server.CreateObject** method to store objects in the **Session** object.

Sending Content to the Browser

As a script in an ASP page is processed, any text or graphics not enclosed within ASP delimiters or <SCRIPT> tags is simply returned to the browser. You can also explicitly send content to the browser by using the **Response** object.

Sending Content

To send content to the browser from within ASP delimiters or from a procedure, use the **Write** method of the **Response** object. For example, the following statement sends a different greeting to the user depending on whether the user has visited the page before:

```
<%
  If blnFirstTime Then
    Response.Write "<H3 ALIGN=CENTER>Welcome to the Overview Page.</H3>"
  Else
    Response.Write "<H3 ALIGN=CENTER>Welcome Back to the Overview Page.</H3>"
  End If
%>
```

Outside of a procedure, you do not have to use **Response.Write** to send content back to the user. Content that is not within scripting delimiters is sent directly to the browser, which formats and displays this content accordingly. For example, the following script produces the same output as the previous script:

```
<H3 ALIGN=CENTER>
<% If blnFirstTime Then %>
  Welcome to the Overview Page.
<% Else %>
  Welcome Back to the Overview Page.
<% End If %>
</H3>
```

Intersperse script commands and HTML when you just need to return output once or when it is more convenient to add statements to existing HTML text. Use **Response.Write** when you do not want to break up a statement with delimiters or when you want to build the string that is returned to the browser. For example, you could construct a string of text that builds a table row with values sent by an HTML form:

```
Response.Write "<TR><TD>" & Request.Form("FirstName") _

 & "</TD><TD>" & Request.Form("LastName") & "</TD></TR>"
```

Request.Form returns the values sent from an HTML form (see Processing User Input).

Note The ampersand character (&) is the string-concatenation character for VBScript. The underscore (_) is the VBScript line-continuation character.

Setting the Content Type

When the Web server returns a file to a browser, it tells the browser what type of content is contained in the file. This enables the browser to determine whether it can display the file itself or whether it has to call another application. For example, if the Web server returns a Microsoft Excel worksheet, the browser must be able to start a copy of Microsoft Excel to display the page. The Web server recognizes file types by mapping the file name extension to a list of *MIME* (Multipurpose Internet Mail Extensions) types. For example, to start Microsoft Excel, the browser needs to recognize the application/vnd.ms-excel MIME type.

You can use the **ContentType** property of the **Response** object to set the HTTP content type string for the content you send to a user. For example, the following command sets the content type for channel definitions:

```
<% Response.ContentType = "application/x-cdf" %>
```

For more information about channels, see Creating Dynamic Channels in this topic.

Other common content types are text/plain (for content returned as text instead of interpreted HTML statements), image/gif (for GIF images), image/jpeg (for JPEG images), video/quicktime (for movies in the Apple QuickTime® format), and text/xml (for XML documents). In addition, a Web server or Web browser may support custom MIME types. To see the content types already defined by your Microsoft Web server, use the Internet Information Services snap-in, to open the property sheets for your Web site, click the **HTTP Headers** tab, and then click the **File Types** tab. These file types may be used as a reference when you choose to manually set the content type with ASP.

Redirecting the Browser

Instead of sending content to a user, you can redirect the browser to another URL with the **Redirect** method. For example, if you want to make sure users have entered your application from a home page so that they receive a customer ID, you can check to see if they have a customer ID; if they do not, you can redirect them to the home page.

```
<%
  If Session("CustomerID") = "" Then
    Response.Redirect "Register.asp"
  End If
%>
```

server-side scripts which are processed before any content is sent to the user are said to be *buffered*. ASP enables you to turn buffering on or off, and this configuration can greatly affect the behavior of the **Redirect** method. Specifically, if you have buffering turned off, then you must redirect the browser before your page's HTTP headers are returned to the browser.

Place the **Response.Redirect** statement at the top of the page, before any text or <HTML> tags, to ensure that nothing has been returned to the browser. If you use **Response.Redirect** after content or headers have been returned to the browser, you will see an error message. Also note that **Response.Redirect** does *not* need to be followed by **Response.End**.

If you want to use **Response.Redirect** from the middle of a page, use it along with the **Response.Buffer** property, as explained in the Buffering Content section in this topic.

Transferring Between .ASP Files

Using **Response.Redirect** to redirect a browser requires a *round-trip*, meaning that the server sends an HTTP response to the browser indicating the location of the new URL. The browser automatically leaves the server's request queue and sends a new HTTP request for this URL. The server then adds this request to the request queue along with other client's requests that arrived in the meantime. For a busy Web site, round-trips can waste bandwidth and reduce server performance—especially when the browser is redirected to a file located on the same server.

You can use the **Server.Transfer** method to transfer from one .asp file to another file located on the same server instead of the **Response.Redirect** method. With **Server.Transfer** you can directly transfer requests for .asp files without ever leaving the server request queue, thus eliminating costly round-trips.

For example, the following script demonstrates how you could use **Server.Transfer** to jump between the pages of an application depending on state information:

```
<%
  If Session("blnSaleCompleted") Then
    Server.Transfer("/Order/ThankYou.asp")
  Else
    Server.Transfer("/Order/MoreInfo.asp")
  End if
%>
```

Server.Transfer sends requests from one executing .asp file to another file. During a transfer, the originally requested .asp file immediately terminates execution without clearing the output buffer (for more information, see the Buffering Content section). Request information is then made available to the destination file when it begins execution. During execution, this file has access to the same set of intrinsic objects (**Request**, **Response**, **Server**, **Session**, and **Application**) as the originally requested file.

You can also use **Server.Transfer** to transfer between .asp files located in different applications. However, when you transfer to an .asp file located in another application, the file will behave as if it was part of the application that initiated the transfer (that is, the file has access only to variables scoped for the initiating application, not for the application where the file actually resides). For example, if you transfer from a file located in the Sales Application to a file located in the Personnel Application, then the Sales Application effectively borrows this file from the Personnel Application and runs it as if it were part of the Sales Application.

ASP also provides the **Server.Execute** command that you can use to transfer to a file, execute its content, and then return to the file that initiated the transfer. If you are familiar with VBScript, it will help you to think of **Server.Execute** as analogous to a procedure call, except that instead of executing a procedure, you are executing an entire .asp file.

For example, the following script demonstrates how you could use **Server.Execute** to do dynamic inclusion of .asp files:

```
<%
  .
  .
  If blnUseDHTML Then
    Server.Execute("DHTML.asp")
  Else
    Server.Execute("HTML.asp")
  End If
  .
  .
%>
```

As long as the destination file belongs to an application on the same server, the originating application will transfer to this file, execute its contents, and then resume executing the file that initiated the transfer. Just as with **Server.Transfer**, an executed .asp file behaves as if it were part of the originating application. **Server.Execute**, however, will *not* work across servers. For more information, see **Server.Execute**.

Buffering Content

By default, the Web server processes all script commands on a page before any content is sent to the user. This process is known as *buffering*. You can use the **Buffer** property of the **Response** object to disable buffering, so that the Web server returns HTML and the results of scripts as it processes a page.

The advantage of buffering your .asp files is that you can abort sending a Web page if, for example, the script processing does not proceed correctly or if a user does not have appropriate security credentials. Instead, you can transfer the user to another page using **Server.Transfer**, or clear the buffer (using the the **Clear** method of the **Response** object) to send different content to the user. Depending on your application, you may want to use the **Clear** method before transferring. The following example uses both of these methods:

```
<HTML>
  <BODY>
  .
  .
  <%
    If Request("CustomerStatus") = "" Then
      Response.Clear
      Server.Transfer("/CustomerInfo/Register.asp")
    Else
      Response.Write "Welcome back " & Request("FirstName") & "!"
                      .
                      .
    End If
  %>
  </BODY>
</HTML>
```

You could also use **Response.Buffer** to prevent the Web server from returning the HTTP header before a script can modify the header. Certain properties and methods, such as **Response.Expires** and **Response.Redirect,** modify the HTTP header.

If the **Buffer** property in a script is set to TRUE without also calling the **Flush** method to immediately send buffered content to the browser, the server will maintain Keep-Alive requests made by the client. The benefit of writing scripts in this manner is that server performance is improved because the server does not have to create a new connection for each client request (assuming that the server, client, and any proxy servers all support Keep-Alive requests). However, a potential drawback to this approach is that buffering prevents the server's response from being sent to the user until the server has finished processing the entire script. For long or complicated scripts, users could experience long wait times before seeing the page.

Buffering is turned on by default for ASP applications. You can use the Internet Information Services snap-in, to turn off buffering for an entire ASP application. For more information, see Configuring ASP Applications.

Allowing Proxy Servers to Cache Pages

Your application may be sending pages to a client through a *proxy server*. A proxy server acts on behalf of client browsers to request pages from Web sites. The proxy server caches HTML pages so that repeated requests for the same page can be returned quickly and efficiently to browsers. Having the proxy server process requests and cache pages reduces the load on the network and on the Web server.

Although caching works well for many HTML pages, it often does not work well for ASP pages that contain frequently updated information. For example, pages that report stock market prices or display inventory for a high-volume business must provide timely information. Information that is even one hour old might not be accurate enough. If your application returns personalized information, such as a custom home page, you want to ensure that no user sees another user's personal information.

By default, ASP instructs proxy servers not to cache the ASP page itself (although images, image maps, applets, and other items referenced from the page are cached). You can allow caching for certain pages by using the **Response.CacheControl** property to set the Cache-Control HTTP header field. The default value of **Response.CacheControl** is the string "Private", which prevents proxy servers from caching the page. To allow caching, set the Cache-Control header field to Public:

```
<% Response.CacheControl = "Public" %>
```

Because HTTP headers must be sent to the browser or proxy before any page content is sent, either put the **Response.CacheControl** property before any HTML tags or, if you have disabled buffering, use **Response.Buffer** to buffer the page.

The Cache-Control header field is part of the HTTP 1.1 specification. ASP pages are not cached on proxy servers that support only HTTP 1.0 because no Expires header field is sent.

Preventing Browsers from Caching Pages

Each browser version has its own rules for determining whether to cache pages. To prevent a browser from caching ASP pages, use **Response.Expires** to set the Expires header:

```
<% Response.Expires = 0 %>
```

A value of 0 forces cached pages to expire immediately. Because HTTP headers must be sent to the browser before any page content is sent, either put the **Response.Expires** property before any HTML tags or buffer the page.

Creating Dynamic Channels

A *channel* is a Web technology available with Microsoft Internet Explorer 4.0, or later, that you can use to automatically deliver new or updated Web content to users. The channel schedules the user's computer to periodically connect to a server and retrieve updated information. (This retrieval process is commonly referred to as *client pull* because information is "pulled" in, or gathered, from the server.) When new information is made available at a particular Web site, the content is downloaded to the browser cache for offline viewing. Clever use of channels for distributing Web based information (especially on intranets) can help to centralize information as well as reduce server traffic. For more information about channels, visit the Microsoft Internet Explorer Web site.

Using ASP, you can write scripts that dynamically create channels by generating a *channel definition file*. An XML-based channel definition file (.cdf) describes the organization and update schedule of a channel's contents. Commands in the .cdf file use syntax similar to HTML tags, so they are easy to learn and to generate from a script. When you write a server-side script to create a channel definition file, give the file a .cdx extension. When ASP reads a file with a .cdx extension, it automatically sends the application/x-cdf content type, which tells the browser to interpret the bytes as channel definitions. If you do not use the .cdx extension, your script must manually set the content type to application/x-cdf by using **Response.ContentType**.

Here is an example of how you might use channels. The following HTML form asks the user to select channels. When submitted, the form calls a script in a .cdx file to create the channel definitions.

```
<P> Choose the channels you want. </P>
<FORM METHOD="POST" ACTION="Chan.cdx">
<P><INPUT TYPE=CHECKBOX NAME=Movies> Movies
<P><INPUT TYPE=CHECKBOX NAME=Sports> Sports
<P><INPUT TYPE="SUBMIT" VALUE="SUBMIT">
</FORM>
```

The script in Chan.cdx builds the channel definitions based on the form values submitted with the request.

```
<% If Request.Form("Movies") <> "" Then %>
  <CHANNEL>
    channel definition statements for the movie pages
  </CHANNEL>
<% End If %>

<% If Request.Form("Sports") <> "" Then %>
  <CHANNEL>
    channel definition statements for the sports pages
  </CHANNEL>
<% End If %>
```

Accessing Server Resources with WebDAV

Distributed Authoring and Versioning (WebDAV), a powerful extension of the HTTP 1.1 protocol, exposes Web file storage media—such as a local file system—over an HTTP connection. WebDAV holds great promise for making the Web into seamless, collaborative authoring environment. With the IIS 5.1 implementation of WebDAV, you can enable remote authors to create, delete, move, search, or apply attributes to files and directories on your Web server. For more information, see WebDav Publishing.

Including Files

Server-side include directives give you a way to insert the content of another file into a file before the Web server processes it. ASP implements only the **#include** directive of this mechanism. To insert a file into an .asp file, use the following syntax:

```
<!-- #include virtual | file ="filename" -->
```

The **virtual** and **file** keywords indicate the type of path you are using to include the file, and *filename* is the path and file name of the file you want to include.

Included files do not require a special file name extension; however, it is considered good programming practice to give included files an .inc extension to distinguish them from other types of files.

Using the Virtual Keyword

Use the **virtual** keyword to indicate a path beginning with a *virtual directory*. For example, if a file named Footer.inc resides in a virtual directory named /Myapp, the following line would insert the contents of Footer.inc into the file containing the line:

```
<!-- #include virtual ="/myapp/footer.inc" -->
```

Using the File Keyword

Use the **file** keyword to indicate a *relative* path. A relative path begins with the directory that contains the including file. For example, if you have a file in the directory Myapp, and the file Header1.inc is in Myapp\Headers, the following line would insert Header1.inc in your file:

```
<!-- #include file ="headers\header1.inc" -->
```

Note that the path to the included file, Headers\header1.inc, is relative to the including file; if the script containing this **#include** statement is not in the directory /Myapp, the statement would not work.

You can also use the **file** keyword with the syntax (`..\`) to include a file from a parent, or higher-level, directory if the **Enable Parent Paths** option is selected in the Internet Information Services snap-in.

For instructions, see Configuring ASP Applications.

Location of Included Files

ASP detects changes to an included file regardless of its location and inserts the files content the next time a browser requests an .asp file which includes this file. However, in general, it is easier to secure include files if they reside within the same application or Web site. For better security, it is advisable to place include files in a separate directory within your application, such as \Includes, and apply only appropriate Execute (Web server) permissions. For more information, see Setting Web Server Permissions.

Important By default, Web server Read permissions are applied to all files. However, to prevent users from viewing the contents of your include files, disable Read permissions for the Include directory.

Including Files: Tips and Cautions

An included file can, in turn, include other files. An .asp file can also include the same file more than once, provided that the **#include** directives do not cause a loop. For example, if the file First.asp includes the file Second.inc, Second.inc must not in turn include First.asp. Nor can a file include itself. ASP detects such loop or nesting errors, generates an error message, and stops processing the requested .asp file.

ASP includes files before executing script commands. Therefore, you cannot use a script command to build the name of an included file. For example, the following script would not open the file Header1.inc because ASP attempts to execute the **#include** directive before it assigns a file name to the variable name.

```
<!--  This script will fail -->
<% name=(header1 & ".inc") %>
<!-- #include file="<%= name %>" -->
```

Scripts commands and procedures must be entirely contained within the script delimiters <% and %>, the HTML tags <SCRIPT> and </SCRIPT>, or the HTML tags <OBJECT> and </OBJECT>. That is, you cannot open a script delimiter in an including .asp file, then close the delimiter in an included file; the script or script command must be a complete unit. For example, the following script would not work:

```
<!-- This script will fail -->
<%
  For i = 1 To n
    statements in main file
    <!--  #include file="header1.inc" -->
  Next
%>
```

The following script, however, would work:

```
<%
  For i = 1 to n
    statements in main file
%>
<!--  #include file="header1.inc"   -->
<% Next %>
```

Note If the file that your ASP script includes contains a large number of functions and variables that are unused by the including script, the extra resources occupied by these unused structures can adversely affect performance, and ultimately decrease the scalability of your Web application. Therefore, it is generally advisable to break your include files into multiple smaller files, and include only those files required by your server-side script, rather than include one or two larger include files that may contain superfluous information.

Occasionally, it may be desirable to include a server-side file by using the HTML <SCRIPT></SCRIPT> tags. For example, the following script includes a file (by means of a relative path) that can be executed by the server:

```
<SCRIPT LANGUAGE="VBScript" RUNAT=SERVER SRC="Utils\datasrt.inc"></SCRIPT>
```

The following table shows the correct syntax for including files with the SRC attribute by means of either virtual or relative paths:

Type of Path	**Syntax**	**Example**
Relative	SRC="*Path\Filename*"	SRC="Utilities\Test.asp"
Virtual	SRC="/*Path/Filename*"	SRC="/MyScripts/Digital.asp"
Virtual	SRC="\Path\Filename"	SRC="\RegApps\Process.asp"

Note You should not put any programmatic logic between the <SCRIPT> tags when including by this method; use another set of <SCRIPT> tags to add such logic.

Managing Sessions

One of the challenges to developing a successful Web application is maintaining user information over the course of a visit, or *session*, as the user travels from page to page in an application. HTTP is a stateless protocol, meaning that your Web server treats each HTTP request for a page as an independent request; the server retains no knowledge of previous requests, even if they occurred only seconds prior to a current request. This inability to remember previous requests means that it is this difficult to write applications, such as an online catalog, where the application may need to track the catalog items a user has selected while jumping between the various pages of the catalog.

ASP provides a unique solution for the problem of managing session information. Using the ASP **Session** object and a special user ID generated by your server, you can create clever applications that identify each visiting user and collect information that your application can then use to track user preferences or selections.

Important ASP assigns the user ID by means of an HTTP cookie, which is a small file stored on the user's browser. So, if you are creating an application for browsers that do not support cookies, or if your customers might set their browsers to refuse cookies, you should not use ASP's session management features.

Starting and Ending Sessions

A session can begin in four ways:

- A new user requests a URL that identifies an .asp file in an application, and the Global.asa file for that application includes a Session_OnStart procedure.
- A user stores a value in the **Session** object.
- A new session automatically starts whenever the server receives a request that does not contain a valid SessionID cookie.
- A user requests an .asp file in an application, and the application's Global.asa file uses the <OBJECT> tag to instantiate an object with session scope. See Using Components and Objects for more information about using the <OBJECT> tag to instantiate an object.

A session automatically ends if a user has not requested or refreshed a page in an application for a specified period of time. This value is 20 minutes by default. You can change the default for an application by setting the **Session Timeout** property on the **Application Options** property sheet in the Internet Information Services snap-in. Set this value according to the requirements of your Web application and the memory capacity of your server. For example, if you expect that users browsing your Web application will linger on each page for only a few minutes, then you may want to significantly reduce the session timeout value from the default. A long session timeout period can result in too many open sessions, which can strain your server's memory resources.

If, for a specific session, you want to set a timeout interval that is shorter than the default application timeout, you can also set the **Timeout** property of the **Session** object. For example, the following script sets a timeout interval of 5 minutes.

```
<%  Session.Timeout = 5  %>
```

You can also set the timeout interval to be greater than the default value, the value determined by the **Session Timeout** property.

Note **Timeout** only applies to sessions that have state. During a *stateless session* the **Session** object does not contain content or static objects. This type of session automatically ends after the request is processed and is recreated on the next request from the same browser.

Alternatively, to deliberately end a session you can use the **Abandon** method of the **Session** object. For example, you can provide a Quit button on a form with the ACTION parameter set to the URL of an .asp file that contains the following command.

```
<% Session.Abandon %>
```

Note A user's requests that are queued up for execution prior to initiating **Session.Abandon** will execute within the context of the session being abandoned. After **Session.Abandon** has completed execution, new incoming requests will not be associated with the session.

About SessionID and Cookies

The first time a user requests an .asp file within a given application, ASP generates a *SessionID*. A number produced by a complex algorithm, the SessionID uniquely identifies each user's session. At the beginning of a new session, the server stores the Session ID in the user's Web browser as a cookie.

The SessionID cookie is similar to a locker key in that, as the user interacts with an application during a session, ASP can store information for the user in a "locker" on the server. The user's SessionID cookie, transmitted in the HTTP request header, enables access to this information in the way that a locker key enables access to a locker's contents. Each time that ASP receives a request for a page, it checks the HTTP request header for a SessionID cookie.

After storing the SessionID cookie in the user's browser, ASP reuses the same cookie to track the session, even if the user requests another .asp file, or requests an .asp file running in other application. Likewise, if the user deliberately abandons or lets the session timeout, and then proceeds to request another .asp file, ASP begins a new session using the same cookie. The only time a user receives a new SessionID cookie is when the server administrator restarts the server, thus clearing the SessionID settings stored in memory, or the user restarts the Web browser.

By reusing the SessionID cookie, ASP minimizes the number of cookies sent to the browser. Additionally, if you determine that your ASP application does not require session management, you can prevent ASP from tracking session and sending SessionID cookies to users.

ASP will not send the session cookies under the following conditions:

- If an application has session state disabled.
- If an ASP page is defined as sessionless, that is, a page containing the

```
<%@ EnableSessionState=False %>
```

tag. For more information, see Sessionless ASP Pages.

You should also note that SessionID cookies are not intended to provide a permanent means for tracking users across multiple visits to a Web site. The SessionID information stored in the server computer's memory can be easily lost. If you want track users who visit your Web application over a longer periods, you must create a user identification by storing a special cookie in a user's Web browser and saving the cookie information to a database. For more information, see Using Cookies.

Storing and Removing Data from the Session object

The **Session** object provides a dynamic, associative array into which you can store information. You can store scalar variables and object variables into the **Session** object.

To store a variable in the **Session** object, assign a value to a named entry in the **Session** object. For example, the following command stores two new variables in the **Session** object:

```
<%
  Session("FirstName") = "Jeff"
  Session("LastName") = "Smith"
%>
```

To retrieve information from the **Session** object, access the named entry. For example, to display the current value of Session("FirstName"):

```
Welcome <%= Session("FirstName") %>
```

You can store user preferences in the **Session** object, and then access that preference to determine what page to return to the user. For example, you can allow a user to specify a text-only version of your content in the first page of the application and apply this choice on all subsequent pages that the user visits in this application.

```
<% If Session("ScreenResolution") = "Low" Then %>
  This is the text version of the page.
<% Else %>
  This is the multimedia version of the page.
<% End If %>
```

You can also store an object instance in the **Session** object, although doing so can affect server performance. For more information, see Setting Object Scope.

At times, it may be desirable to delete items stored in the **Session** object. For example, it is not uncommon for users visiting an online retail store to change their minds, abandon a list of purchase items, and decide on a completely different set of selections. In such a case it may be expedient to update the **Session** object by deleting inappropriate values.

The **Session** object's **Contents** collection contains all of the variables that have been stored (that is, those stored without using the HTML <OBJECT> tag) for a session. By using the **Contents** collection's **Remove** method, you can selectively remove a reference to a variable that was added for the session state. The following script illustrates how to use the **Remove** method to purge an item, in this case user discount information, from the **Session** object:

```
<%
  If Session.Contents("Purchamnt") <= 75 then
    Session.Contents.Remove("Discount")
  End If
%>
```

If desirable, you can also use the **Contents** collection's **RemoveAll** method to completely remove all variables stored for the session:

```
Session.Content.RemoveAll()
```

Using the **Remove** method you can choose to delete items by name or by index. The following script demonstrates how to cycle through values stored in the **Session** object and conditionally remove values by index:

```
<%
  For Each intQuote in Session.Contents
    If Session.Contents(intQuote) < 200 Then
      Session.Contents.Remove(intQuote)
    End If
  Next
%>
```

Managing Sessions Across Multiple Servers

ASP session information is stored on the Web server. A browser must request pages from the same Web server for scripts to access session information. On cluster of Web servers (where many Web servers share the responsibility for responding to user requests) user requests will not always be routed to the same server. Instead, special software distributes all requests for the site URL to whichever server is free, a process called *load balancing*. Load balancing makes it difficult to maintain session information on a cluster of Web servers.

To use ASP session management on a load-balanced site, you must ensure that all requests within a user session are directed to the same Web server. One way to do this is to write a **Session_OnStart** procedure that uses the **Response** object to redirect the browser to the specific Web server on which the user's session is running. If all links in your application pages are relative, future requests for a page will be routed to the same server.

For example, a user might access an application by requesting the general URL for a site: http://www.microsoft.com. The load balancer routes the request to a specific server, for example, server3.microsoft.com. ASP creates a new user session on that server. In the **Session_OnStart** procedure, the browser is redirected to the specified server:

```
<% Response.Redirect("http://server3.microsoft.com/webapps/firstpage.asp") %>
```

The browser will request the specified page, and all subsequent requests will be routed to the same server as long as specific server names are not referenced in the original URLs.

Using Cookies

A cookie is a token that the Web server embeds in a user's Web browser to identify the user. The next time the same browser requests a page, it sends the cookie it received from the Web server. Cookies allow a set of information to be associated with a user. ASP scripts can both get and set the values of cookies by using the **Cookies** collection of the **Response** and **Request** objects.

Setting Cookies

To set the value of a cookie, use **Response.Cookies**. If the cookie does not already exist, **Response.Cookies** creates a new one. For example, to send a cookie named ("VisitorID") with an associated value ("49") to the browser, use the following command, which must appear on your Web page before the <HTML> tag:

```
<% Response.Cookies("VisitorID") = 49 %>
```

If you want a cookie to be used only during the current user session, then sending the cookie to the browser is all you need to do. However, if you want to identify a user even after the user has stopped and restarted the browser, you must force the browser to store the cookie in a file on the client computer's hard disk. To save the cookie, use the **Expires** attribute for **Response.Cookies** and set the date to some date in the future:

```
<%
  Response.Cookies("VisitorID") = 49
  Response.Cookies("VisitorID").Expires = "December 31, 2001"
%>
```

A cookie can have multiple values; such a cookie is called an *indexed cookie*. An indexed cookie value is assigned a key; you can set a particular cookie key value. For example:

```
<% Response.Cookies("VisitorID")("49") = "Travel" %>
```

If an existing cookie has key values but **Response.Cookies** does not specify a key name, then the existing key values are deleted. Similarly, if an existing cookie does not have key values but **Response.Cookies** specifies key names and values, the existing value of the cookie is deleted and new key-value pairs are created.

Getting Cookies

To get the value of a cookie, use the **Request.Cookies** collection. For example, if the user HTTP request sets `VisitorID=49`, then the following statement retrieves the value `49`:

```
<%= Request.Cookies("VisitorID") %>
```

Similarly, to retrieve a key value from an indexed cookie, use the key name. For example, if a user's browser sends the following information in the HTTP request header:

```
Cookie: VisitorID=49=Travel
```

The following statement would then return the value `Travel`:

```
<%= Request.Cookies("VisitorID")("49") %>
```

Setting Cookie Paths

Each cookie stored by ASP on the user's Web browser contains path information. When the browser requests a file stored in the same location as the path specified in the cookie, the browser automatically forwards the cookie to the server. By default, cookie paths correspond to the name of the application containing the .asp file that originally generated the cookie. For example, if an .asp file, residing in an application called *UserApplication*, generates a cookie, then each time a user's Web browser retrieves any file residing in that application, the browser will forward the cookie, in addition to any other cookies containing the path */UserApplication*.

To specify a path for a cookie other than the default application path, you can use the ASP **Response.Cookies** collection's **Path** attribute. For example, the following script assigns the path SalesApp/Customer/Profiles/ to a cookie called `Purchases`:

```
<%
  Response.Cookies("Purchases") = "12"
  Response.Cookies("Purchases").Expires = "January 1, 2001"
  Response.Cookies("Purchases").Path = "/SalesApp/Customer/Profiles/"
%>
```

Whenever the Web browser containing the `Purchases` cookie requests a file residing in the path /SalesApp/Customer/Profiles/ or in any of it subdirectories, the browser forwards the cookie to the server.

Many Web browsers, including Microsoft Internet Explorer version 4.0, or later, and Netscape browsers, preserve the case of the cookie path. This means that if the case of the path of a requested file differs from the case of the stored cookie path, the browser will not send the cookie to the server. For example, to ASP, the virtual directories /TRAVEL and /travel are the same ASP application, but to a browser that preserves the case of a URL, /TRAVEL and /travel are two different applications. Make sure all URLs to .asp files have the same case to ensure that the user's browser forwards stored cookies.

You can use the following statement to set the cookie path so that the user's Web browser will forward a cookie whenever the browser requests a file from your server, regardless of application or path:

```
Response.Cookies("Purchases").Path = "/"
```

Note, however, that forwarding cookies to the server, without distinguishing between applications, raises a potential security concern if the cookies contain sensitive information that should not be accessible outside of a specific application.

Preserving State without Cookies

Not all browsers support cookies. Even with browsers that do support cookies, some users prefer to turn off cookie support. If your application needs to be responsive to browsers that don't support cookies, you cannot use ASP session management.

In this case, you must write your own mechanism to pass information from page to page in your application. There are two general ways to do this:

- Add parameters to a URL's query string. For example:

  ```
  http://MyServer/MyApp/start.asp?name=Jeff
  ```

  Some browsers, however, will discard any explicit parameters passed in a query string if a form is submitted with the GET method.

- Add hidden values to a form. For example, the following HTML form contains a hidden control, which does not appear on the actual form and remains invisible in the user's Web browser. The form passes a user identification value, in addition to the information supplied by the user, by using the HTTP POST method.

- `<FORM METHOD="POST" ACTION="/scripts/inform.asp">`
- `<INPUT TYPE="text" NAME="city" VALUE="">`
- `<INPUT TYPE="text" NAME="country/region" VALUE ="">`
- `<INPUT TYPE="hidden" NAME="userid" VALUE= <%= UserIDNum(i) %>`

  `<INPUT TYPE="submit"  VALUE="Enter">`

  This method requires all link destinations that pass user information to be coded as HTML forms.

If you are not using ASP session management, you should turn off session support for your application. When sessions are enabled, ASP sends a SessionID cookie to each browser that requests a page. To turn off session support, clear the **Enable Session State** check box on the **Application Options** property sheet in the Internet Information Services snap-in.

Sessionless ASP Pages

With ASP, you can also create sessionless pages which can be used to put off the creation of sessions tracking until needed.

Sessionless pages do *not* carry out the following:

- Execute **Session_OnStart** procedures.
- Send session ID cookies.
- Create **Session** objects.
- Access built-in **Session** objects or session scope objects created with the <OBJECT> tag.
- Serialize execution with other session requests.

To configure an .asp file as sessionless, use the following:

```
<%@ EnableSessionState=False %>
```

You should place this script as the first line in your .asp file, before any other scripts. The default, when this tag is omitted, enables session tracking.

Sessionless ASP pages can often improve the responsiveness of your server by eliminating potentially time consuming session activity. For example, consider the case of an ASP page containing two HTML frames: frames 1 and 2, both within one frameset. Frame 1 contains an .asp file that executes a complex script, while frame 2 contains a simpler .asp file. Because ASP executes session requests in sequential order, or *serially*, you will not be able to see the contents of frame 2 until the script in frame 1 has executed. However, if you make the .asp file for frame 1 sessionless, then ASP requests will no longer be serialized and the browser will render the contents of frame 2 before the contents of frame 1 have finished executing.

Unfortunately, the way in which multiple requests for different frames are processed ultimately depends on the configuration of the user's Web browser. Certain Web browsers may serialize requests despite the sessionless configuration of your .asp files.

Accessing a Data Source

ActiveX Data Objects (ADO) are an easy-to-use yet extensible technology for adding database access to your Web pages. You can use ADO to write compact and scalable scripts for connecting to OLE DB compliant data sources, such as databases, spreadsheets, sequential data files, or e-mail directories. OLE DB is a system-level programming interface that provides standard set of COM interfaces for exposing database management system functionality. With ADO's object model you can easily access these interfaces (using scripting languages, such as VBScript or JScript) to add database functionality to your Web applications. In addition, you can also use ADO to access Open Database Connectivity (ODBC) compliant databases.

If you are a scripter with a modest understanding of database connectivity, you will find ADO's command syntax uncomplicated and easy-to-use. If you are an experienced developer you will appreciate the scalable, high-performance access ADO provides to a variety of data sources.

For more information about ADO, visit the Microsoft Universal Data Access (UDA) Web site.

Creating a Connection String

The first step in creating a Web data application is to provide a way for ADO to locate and identify your data source. This is accomplished by means of a *connection string*, a series of semicolon delimited arguments that define parameters such as the data source provider and the location of the data source. ADO uses the connection string to identify the OLE DB *provider* and to direct the provider to the data source. The provider is a component that represents the data source and exposes information to your application in the form of rowsets.

The following table lists OLE DB connection strings for several common data sources:

Data Source	**OLE DB Connection String**
Microsoft® Access	Provider=Microsoft.Jet.OLEDB.4.0;Data Source=*physical path to .mdb file*
Microsoft SQL Server	Provider=SQLOLEDB.1;Data Source=*path to database on server*
Oracle	Provider=MSDAORA.1;Data Source=*path to database on server*
Microsoft Indexing Service	Provider=MSIDXS.1;Data Source=*path to file*

To provide for backward compatibility, the OLE DB Provider for ODBC supports ODBC connection string syntax. The following table lists commonly used ODBC connection strings:

Data Source Driver	**ODBC Connection String**
Microsoft Access	Driver={Microsoft Access Driver (*.mdb)};DBQ=*physical path to .mdb file*
SQL Server	DRIVER={SQL Server};SERVER=*path to server*

Oracle	DRIVER={Microsoft ODBC for Oracle};SERVER=*path to server*
Microsoft Excel	Driver={Microsoft Excel Driver (*.xls)};DBQ=*physical path to .xls file*; DriverID=278
Microsoft Excel 97	Driver={Microsoft Excel Driver (*.xls)};DBQ=*physical path to .xls file*;DriverID=790
Paradox	Driver={Microsoft Paradox Driver (*.db)};DBQ=*physical path to .db file*;DriverID=26
Text	Driver={Microsoft Text Driver (*.txt;*.csv)};DefaultDir=*physical path to .txt file*
Microsoft Visual FoxPro® (with a database container)	Driver={Microsoft Visual FoxPro Driver};SourceType=DBC;SourceDb=*physical path to .dbc file*
Microsoft Visual FoxPro (without a database container)	Driver={Microsoft Visual FoxPro Driver};SourceType=DBF;SourceDb=*physical path to .dbf file*

Note Connection strings that use a UNC path to refer to a data source located on a remote computer can pose a potential security issue. To prevent unauthorized access of your data source, create a Windows account for computers requiring access to the data and then apply appropriate NTFS permissions to the data source. For more information, see Securing Your Files with NTFS.

Advanced Issues to Consider When Designing Web Data Applications

For performance and reliability reasons, it is strongly recommended that you use a client-server database engine for the deployment of data driven Web applications that require high-demand access from more than approximately 10 concurrent users. Although ADO works with any OLE DB compliant data source, it has been extensively tested and is designed to work with client server databases such as Microsoft SQL Server or Oracle.

ASP supports shared file databases (Microsoft Access or Microsoft FoxPro) as valid data sources. Although some examples in the ASP documentation use a *shared file* database, it is recommended that these types of database engines be used only for development purposes or limited deployment scenarios. Shared file databases may not be as well suited as client-server databases for very high-demand, production-quality Web applications.

If you are developing an ASP database application intended to connect to a remote SQL Server database you should also be aware of the following issues:

- **Choosing Connection Scheme for SQL Server** You can choose between the TCP/IP Sockets and Named Pipes methods for accessing a remote SQL Server database. With Named Pipes, database clients must be authenticated by Windows before establishing a connection, raising the possibility that a remote computer running named pipes might deny access to a user who has the appropriate SQL Server access credentials, but does not have a Windows user account on that computer. Alternatively, connections using TCP/IP Sockets connect directly to the database server, without connecting through an intermediary computer—as is the case with Named Pipes. And because connections made with TCP/IP Sockets connect directly to the database server, users can gain access through SQL Server authentication, rather than Windows authentication.

- **ODBC 80004005 Error** If the connection scheme for accessing SQL Server is not set correctly, users viewing your database application may receive an ODBC 80004005 error message. To correct this situation, try using a local named pipe connection instead of a network named pipe connection if SQL Server is running on the same computer as IIS. Windows XP security rules will not be enforced because the pipe is a local connection rather than a network connection, which can be impersonated by the anonymous user account. Also, in the SQL Server connection string (either in the Global.asa file or in a page-level script), change the parameter **SERVER=*server name*** to **SERVER=(local)**. The keyword (local) is a special parameter recognized by the SQL Server ODBC driver. If this solution does not work, then try to use a non-authenticated protocol between IIS and SQL Server, such as TCP/IP sockets. This protocol will work when SQL Server is running locally or on remote computer.

  Note To improve performance when connecting to a remote databases, use TCP/IP Sockets.

- **SQL Server Security** If you use SQL Server's *Integrated* or *Mixed* security features, and the SQL Server database resides on a remote server, you will not be able to use integrated Windows authentication. Specifically, you cannot forward integrated Windows authentication credentials to the remote computer. This means that you may have to use Basic authentication, which relies on the user to provide user name and password information.

For more information about these issues, visit the Microsoft Product Support Services Web site.

Connecting to a Data Source

ADO provides the **Connection** object for establishing and managing connections between your applications and OLE DB compliant data sources or ODBC compliant databases. The **Connection** object features properties and methods you can use to open and close database connections, and to issue queries for updating information.

To establish a database connection, you first create an instance of the **Connection** object. For example, the following script instantiates the **Connection** object and proceeds to open a connection:

```
<%
  'Create a connection object.
  Set cnn = Server.CreateObject("ADODB.Connection")
  'Open a connection using the OLE DB connection string.
  cnn.Open  "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\MarketData\ProjectedSales.mdb"
%>
```

Note The connection string does not contain spaces before or after the equal sign (=).

In this case, the **Connection** object's **Open** method refers to the connection string.

Executing SQL Queries with the Connection Object

With the **Execute** method of the **Connection** object you can issue commands to the data sources, such as Structured Query Language (SQL) queries. (SQL, an industry standard language for communicating with databases, defines commands for retrieving and updating information.) The **Execute** method can accept parameters that specify the command (or query), the number of data records affected, and the type of command being used.

The following script uses the **Execute** method to issue a query in the form of a SQL **INSERT** command, which inserts data into a specific database table. In this case, the script block inserts the name *Jose Lugo* into a database table named *Customers*.

```
<%
  'Define the OLE DB connection string.
  strConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Data\Employees.mdb"
```

```
  'Instantiate the Connection object and open a database connection.
  Set cnn = Server.CreateObject("ADODB.Connection")
  cnn.Open strConnectionString

  'Define SQL SELECT statement.
  strSQL = "INSERT INTO Customers (FirstName, LastName) VALUES ('Jose','Lugo')"

  'Use the Execute method to issue a SQL query to database.
  cnn.Execute strSQL,,adCmdText + adExecuteNoRecords
%>
```

Note that two parameters are specified in the statement used to execute the query: **adCmdText** and **adExecuteNoRecords**. The optional **adCmdText** parameter specifies the type of command, indicating that the provider should evaluate the query statement (in this case, a SQL query) as a textual definition of a command. The **adExecuteNoRecords** parameter instructs ADO to not create a set of data records if there are no results returned to the application. This parameter works only with command types defined as a text definition, such as SQL queries, or stored database procedures. Although the **adCmdText** and **adExecuteNoRecords** parameters are optional, you should specify theses parameters when using the **Execute** method to improve the performance of your data application.

Important ADO parameters, such as **adCmdText**, need to be defined before you can use them in a script. A convenient way to define parameters is to use a *component type library*, which is a file containing definitions for all ADO parameters. To implement a component type library, it must first be declared. Add the following the <METADATA> tag to your .asp file or Global.asa file to declare the ADO type library:

```
<!--METADATA NAME="Microsoft ActiveX Data Objects 2.5 Library" TYPE="TypeLib"
UUID="{00000205-0000-0010-8000-00AA006D2EA4}"-->
```

For details about implementing component type libraries, see the Using Constants section of the Using Variables and Constants topic.

In addition to the SQL **INSERT** command, you can use the SQL **UPDATE** and **DELETE** commands to change and remove database information.

With the SQL **UPDATE** command you can change the values of items in a database table. The following script uses the **UPDATE** command to change the `Customers` table's `FirstName` fields to `Jeff` for every `LastName` field containing the last name `Smith`.

```
<%
  Set cnn = Server.CreateObject("ADODB.Connection")
  cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Data\Employees.mdb"
  cnn.Execute "UPDATE Customers SET FirstName = 'Jeff' WHERE LastName = 'Smith'
",,adCmdText + adExecuteNoRecords
%>
```

To remove specific records from a database table, use the SQL **DELETE** command. The following script removes all rows from the Customers table where the last name is `Smith`:

```
<%
  Set cnn = Server.CreateObject("ADODB.Connection")
  cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Data\Employees.mdb"
  cnn.Execute "DELETE FROM Customers WHERE LastName = 'Smith'",,adCmdText +
adExecuteNoRecords
%>
```

Note You must be careful when using the SQL **DELETE** command. A **DELETE** command without an accompanying **WHERE** clause will delete all rows from a table. Be sure to include a SQL **WHERE** clause, which specifies the exact rows to be deleted.

Using the Recordset Object for Manipulating Results

For retrieving data, examining results, and making changes to your database, ADO provides the **Recordset** object. As its name implies, the **Recordset** object has features that you can use, depending on your query constraints, for retrieving and displaying a set of database rows, or *records*. The **Recordset** object maintains the position of each record returned by a query, thus enabling you to step through results one item at a time.

Retrieving a Record Set

Successful Web data applications employ both the **Connection** object, to establish a link, and the **Recordset** object, to manipulate returned data. By combining the specialized functions of both objects you can develop database applications to carry out almost any data handling task. For example, the following server-side script uses the **Recordset** object to execute a SQL **SELECT** command. The **SELECT** command retrieves a specific set of information based on query constraints. The query also contains a SQL **WHERE** clause, used to narrow down a query to a specific criterion. In this example, the **WHERE** clause limits the query to all records containing the last name *Smith* from the *Customers* database table.

```
<%
  'Establish a connection with data source.
  strConnectionString  = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Data\Employees.mdb"
  Set cnn = Server.CreateObject("ADODB.Connection")
  cnn.Open strConnectionString

  'Instantiate a Recordset object.
  Set rstCustomers = Server.CreateObject("ADODB.Recordset")

  'Open a recordset using the Open method
  'and use the connection established by the Connection object.
  strSQL = "SELECT FirstName, LastName FROM Customers WHERE LastName = 'Smith' "
  rstCustomers.Open  strSQL, cnn

  'Cycle through record set and display the results
  'and increment record position with MoveNext method.
   Set objFirstName = rstCustomers("FirstName")
   Set objLastName = rstCustomers("LastName")
   Do Until rstCustomers.EOF
     Response.Write objFirstName & " " & objLastName & "<BR>"
     rstCustomers.MoveNext
   Loop

%>
```

Note that in the previous example, the **Connection** object established the database connection, and the **Recordset** object used the same connection to retrieve results from the database. This method is advantageous when you need to precisely configure the way in which the link with the database is established. For example, if you needed to specify the time delay before a connection attempt aborts, you would need to use the **Connection** object to set this property. However, if you just wanted to establish a connection using ADO's default connection properties, you could use **Recordset** object's **Open** method to establish a link:

```
<%
  strConnectionString  = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Data\Employees.mdb"
  strSQL = "SELECT FirstName, LastName FROM Customers WHERE LastName = 'Smith' "
  Set rstCustomers = Server.CreateObject("ADODB.Recordset")

  'Open a connection using the Open method
  'and use the connection established by the Connection object.
  rstCustomers.Open  strSQL, strConnectionString

  'Cycle through the record set, display the results,
  'and increment record position with MoveNext method.
   Set objFirstName = rstCustomers("FirstName")
   Set objLastName = rstCustomers("LastName")
   Do Until rstCustomers.EOF
```

```
      Response.Write objFirstName & " " & objLastName & "<BR>"
      rstCustomers.MoveNext
   Loop
%>
```

When you establish a connection using the **Recordset** object's **Open** method to establish a connection, you are implicitly using the **Connection** object to secure the link. For more information, see Microsoft ActiveX Data Objects (ADO) documentation available from the Microsoft Universal Data Access Web site.

Note To significantly improve the performance of your ASP database applications, consider caching the recordset in **Application** state. For more information, see Caching Data.

It is often useful to count the number of records returned in a recordset. The **Open** method of the **Recordset** object enables you to specify an optional *cursor* parameter that determines how the underlying provider retrieves and navigates the recordset. By adding the **adOpenKeyset** cursor parameter to the statement used to execute the query, you enable the client application to fully navigate the recordset. As a result, the application can use the **RecordCount** property to accurately count the number of records in the recordset. See the following example:

```
<%
         Set rs = Server.CreateObject("ADODB.Recordset")
         rs.Open "SELECT * FROM NewOrders", "Provider=Microsoft.Jet.OLEDB.3.51;Data
Source='C:\CustomerOrders\Orders.mdb'", adOpenKeyset, adLockOptimistic, adCmdText

         'Use the RecordCount property of the Recordset object to get count.
         If rs.RecordCount >= 5 then
           Response.Write "We've received the following " & rs.RecordCount & " new
orders<BR>"

           Do Until rs.EOF
                  Response.Write rs("CustomerFirstName") & " " & rs("CustomerLastName") &
"<BR>"
                  Response.Write rs("AccountNumber") & "<BR>"
                  Response.Write rs("Quantity") & "<BR>"
                  Response.Write rs("DeliveryDate") & "<BR><BR>"
                  rs.MoveNext
           Loop

         Else
           Response.Write "There are less than " & rs.RecordCount & " new orders."
         End If

   rs.Close
%>
```

Improving Queries with the Command Object

With the ADO **Command** object you can execute queries in the same way as queries executed with the **Connection** and **Recordset** object, except that with the **Command** object you can prepare, or compile, your query on the database source and then repeatedly reissue the query with a different set of values. The benefit of compiling queries in this manner is that you can vastly reduce the time required to reissue modifications to an existing query. In addition, you can leave your SQL queries partially undefined, with the option of altering portions of your queries just prior to execution.

The **Command** object's **Parameters** collection saves you the trouble of reconstructing your query each time you want to reissue your query. For example, if you need to regularly update supply and cost information in your Web-based inventory system, you can predefine your query in the following way:

```
<%
    'Open a connection using Connection object. Notice that the Command object
    'does not have an Open method for establishing a connection.
    strConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Data\Inventory.mdb"
```

```
    Set cnn = Server.CreateObject("ADODB.Connection")
    cnn.Open strConnectionString

    'Instantiate Command object; use ActiveConnection property to attach
    'connection to Command object.
    Set cmn= Server.CreateObject("ADODB.Command")
    Set cmn.ActiveConnection = cnn

    'Define SQL query.
    cmn.CommandText = "INSERT INTO Inventory (Material, Quantity) VALUES (?, ?)"

    'Save a prepared (or pre-compiled) version of the query specified in CommandText
    'property before a Command object's first execution.
    cmn.Prepared = True

    'Define query parameter configuration information.
    cmn.Parameters.Append cmn.CreateParameter("material_type",adVarChar, ,255 )
    cmn.Parameters.Append cmn.CreateParameter("quantity",adVarChar, ,255 )

    'Define and execute first insert.
    cmn("material_type") = "light bulbs"
    cmn("quantity") = "40"
    cmn.Execute ,,adCmdText + adExecuteNoRecords

    'Define and execute second insert.
    cmn("material_type") = "fuses"
    cmn("quantity") = "600"
    cmn.Execute ,,adCmdText + adExecuteNoRecords
    .
    .
    .
  %>
```

Important ADO parameters, such as **adCmdText**, are simply variables, this means that before using an ADO parameter in a data access script you need to define its value. Since ADO uses a large number of parameters, it is easier to define parameters by means of a *component type library*, a file containing definitions for every ADO parameter and constant. For details about implementing ADO's type library, see the Using Constants section of the Using Variables and Constants topic.

In the previous example, you will note that the script repeatedly constructs and reissues a SQL query with different values, without having to redefine and resend the query to the database source. Compiling your queries with the **Command** object also offers you the advantage of avoiding problems that can arise from concatenating strings and variables to form SQL queries. In particular, by using the **Command** object's **Parameter** collection, you can avoid problems related to defining certain types of string, date, and time variables. For example, SQL query values containing apostrophes (') can cause a query to fail:

```
  strSQL = "INSERT INTO Customers (FirstName, LastName) VALUES ('Robert','O'Hara')"
```

Note that the last name **O'Hara** contains an apostrophe, which conflicts with the apostrophes used to denote data in the SQL **VALUES** keyword. By binding the query value as a **Command** object parameter, you avoid this type of problem.

Combining HTML Forms and Database Access

Web pages containing HTML forms can enable users to remotely query a database and retrieve specific information. With ADO you can create surprisingly simple scripts that collect user form information, create a custom database query, and return information to the user. Using the ASP **Request** object, you can retrieve information entered into an HTML form and incorporate this information into your SQL statements. For example, the following script block inserts information supplied by an HTML form into a table. The script collects the user information with the **Request** object 's **Form** collection.

```
<%
  'Open a connection using Connection object. The Command object
  'does not have an Open method for establishing a connection.
   strConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\CompanyCatalog\Seeds.mdb"
         Set cnn = Server.CreateObject("ADODB.Connection")
         cnn.Open strConnectionString

  'Instantiate Command object
  'and  use ActiveConnection property to attach
  'connection to Command object.
  Set cmn= Server.CreateObject("ADODB.Command")
  Set cmn.ActiveConnection = cnn

  'Define SQL query.
  cmn.CommandText = "INSERT INTO MySeedsTable (Type) VALUES (?)"

  'Define query parameter configuration information.
  cmn.Parameters.Append cmn.CreateParameter("type",adVarChar, ,255)

  'Assign input value and execute update.
  cmn("type") = Request.Form("SeedType")
  cmn.Execute ,,adCmdText + adExecuteNoRecords
%>
```

For more information about forms and using the ASP **Request** object, see Processing User Input.

Managing Database Connections

One of the main challenges of designing a sophisticated Web database application, such as an online order entry application that services thousands of customers, is properly managing database connections. Opening and maintaining database connections, even when no information is being transmitted, can severely strain a database server's resources and result in connectivity problems. Well designed Web database applications recycle database connections and compensate for delays due to network traffic.

Timing Out a Connection

A database server experiencing a sudden increase in activity can become backlogged, greatly increasing the time required to establish a database connection. As a result, excessive connection delays can reduce the performance of your database application.

With the **Connection** object's **ConnectionTimeout** you can limit the amount of time that your application waits before abandoning a connection attempt and issuing an error message. For example, the following script sets the **ConnectionTimeout** property to wait twenty seconds before cancelling the connection attempt:

```
Set cnn = Server.CreateObject("ADODB.Connection")

cnn.ConnectionTimeout = 20

cnn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Data\Inventory.mdb"
```

The default for the **ConnectionTimeout** property is 30 seconds.

Note Before incorporating the **ConnectionTimeout** property into your database applications, make sure that your connection provider and data source support this property.

Pooling Connections

Connection pooling enables your Web application to use a connection from a *pool*, or reservoir of free connections that do not need to be reestablished. After a connection has been created and placed in a pool, your application reuses that connection without having to perform the connection process. This can result in significant performance gains, especially if your application connects over a network or repeatedly connects and disconnects. In addition, a pooled connection can be used repeatedly by multiple applications.

OLE DB Session Pooling

OLE DB has a pooling feature, called *session pooling*, optimized for improving connectivity performance in large Web database applications. Session pooling preserves connection security and other properties. A pooled connection is only reused if matching requests are made from both sides of the connection. By default, the OLE DB providers for Microsoft SQL server and Oracle support session pooling. This means that you do not have to configure your application, server, or database to use session pooling. However, if your provider does not support session pooling by default, you need to create a registry setting to enable session pooling. For more information about session pooling, see the OLE DB 2.0 Software Development Kit (SDK) documentation.

ODBC Connection Pooling

If you want your ODBC driver to participate in connection pooling you must configure your specific database driver and then set the driver's **CPTimeout** property in the Windows registry. The **CPTimeout** property determines the length of time that a connection remains in the connection pool. If the connection remains in the pool longer than the duration set by **CPTimeout**, the connection is closed and removed from the pool. The default value for **CPTimeout** is 60 seconds.

You can selectively set the **CPTimeout** property to enable connection pooling for a specific ODBC database driver by creating a registry key with the following settings:

```
\HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\driver-name\CPTimeout = timeout

 (REG_SZ, units are in seconds)
```

For example, the following key sets the connection pool timeout to 180 seconds (3 minutes) for the SQL Server driver.

```
\HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\SQL Server\CPTimeout = 180
```

Note By default, your Web server activates connection pooling for SQL Server by setting **CPTimeout** to 60 seconds.

Using Connections Across Multiple Pages

Although you can reuse a connection across multiple pages by storing the connection in ASP's **Application** object, doing so may unnecessarily keep open a connection open, defeating the advantages of using connection pooling. If you have many users that need to connect to the same ASP database application, a better approach is to reuse a database connection string across several Web pages by placing the string in ASP's **Application** object. For example, you can specify a connection string in the Global.asa file's Application_OnStart event procedure, as in the following script:

```
Application("ConnectionString") = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Data\Inventory.mdb"
```

Then in each ASP file that accesses the database, you can write

```
<OBJECT RUNAT=SERVER ID=cnn PROGID="ADODB.Connection"></OBJECT>
```

to create an instance of the connection object for the page, and use the script

```
cnn.Open Application("ConnectionString")
```

to open the connection. At the end of the page, you close the connection with

```
cnn.Close
```

In the case of an individual user who needs to reuse a connection across multiple Web pages, you may find it more advantageous to use the **Session** object rather than the **Application** object for storing the connection string.

Closing Connections

To make the best use of connection pooling, explicitly close database connections as soon as possible. By default, a connection terminates after your script finishes execution. However, by explicitly closing a connection in your script after it is no longer needed, you reduce demand on the database server and make the connection available to other users.

You can use **Connection** object's **Close** method to explicitly terminate a connection between the **Connection** object and the database. The following script opens and closes a connection:

```
<%

  strConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Data\Inventory.mdb"

  Set cnn = Server.CreateObject("ADODB.Connection")

  cnn.Open strConnectionString

  cnn.Close

%>
```

Understanding Transactions

Business applications frequently need to be able to run scripts and components within transactions. A *transaction* is a server operation that succeeds or fails as a whole, even if the operation involves many steps (for example, ordering, checking inventory, and billing). You can create server-side scripts that run within a transaction so that if any portion of the script fails, the entire transaction is aborted.

ASP transaction processing is based on Component Services transactioning environment, a transaction processing system for developing, deploying, and managing high performance, scalable, and robust enterprise, Internet, and intranet server applications. This transactioning environment defines an application programming model for developing distributed, component-based applications. It also provides a run-time environment for deploying and managing these applications.

The functionality required to create transactional scripts is built into your Web server. If you install Component Services, you can also package components so they run within transactions.

About Transactions

A *transaction* is an operation that succeeds or fails as a whole. Transaction processing is used to update databases reliably. When you are making many related changes to a database or updating several databases at once, you want to ensure that all of the changes are correctly executed. If any of the changes fail, you want to restore the original state of the database tables.

Without Component Services, you would have to write your scripts and components to manually track the requested changes and restore data if any changes failed. With Component Services, you simply declare your scripts and components to require transactions and let Component Services handle the transaction coordination. Transaction processing applies only to database access; Component Services cannot roll back changes to the file system or changes to other, nontransactional resources. The database your application accesses must be supported by Component Services. Currently Component Services supports SQL Server and any database that supports the XA protocol from the X/Open consortium. Component Services will continue to expand its support for other databases in the future.

Using the **Server.Transfer** and **Server.Execute** methods a transaction can span multiple ASP pages. If a script contains the @TRANSACTION directive, with the value specified as Required, and the script is called by either the **Server.Transfer** or **Server.Execute** method, then the script will continue the transaction of the calling .asp file if the calling .asp file was transacted. If the calling .asp file was not transacted, the called .asp file will then automatically create a new transaction.

For example, the following script initiates a transaction:

```
<%@ TRANSACTION=Required %>

<%
  .
  .
  'End transaction.
  Server.Transfer("/BookSales/EndTrans.asp")
%>
```

However, following script calls another script that also initializes a transaction:

```
<%@ TRANSACTION=Required%>

<%
  'Instantiate a custom component to close transactions.
  Set objSale = Server.CreateObject("SalesTransacted.Complete")
  .
  .
%>
```

However, the interaction between the two scripts would constitute only a single transaction. For more information about writing scripts with **Server.Transfer** and **Server.Execute**, see Sending Content to the Browser.

Declaring a Transactional Script

When you declare a page to be transactional, any script commands and objects used on the page are run under the same transaction context. Component Services handles the details of creating the transaction and determining whether the transaction succeeds (commits) or fails (aborts). To declare a page transactional, add the @TRANSACTION directive to the top of the page:

```
<%@ TRANSACTION = value %>
```

For more information on the *value* argument, see the @TRANSACTION directive reference.

The @TRANSACTION directive must be the very first line on the page, otherwise an error is generated. You must add the directive to each page that should be run under a transaction. The current transaction ends when the script finishes processing.

Most applications only require transaction context for certain operations. For example, an airline site might use transactional scripts for ticket purchasing and seat assignments. All other scripts could be safely run without a transaction context. Because transactions should be used only for pages that need transaction processing, you cannot declare an application's Global.asa file as transactional.

If a transaction is aborted, Component Services rolls back any changes made to resources that support transactions. Currently, only database servers fully support transactions because this data is the most critical to enterprise applications. Component Services does not roll back changes to files on a hard disk, ASP session and application variables, or collections. You can, however, write scripts that restore variables and collections by writing transaction events, as described later in this topic. Your script can also explicitly commit or abort a transaction if an operation such as writing data to a file fails.

Committing or Aborting a Script

Because Component Services tracks transaction processing, it determines whether a transaction has completed successfully or failed. A script can explicitly declare that it is aborting a transaction by calling **ObjectContext.SetAbort**. For example, your script might abort a transaction if it receives an error from a component, if a business rule is violated (for example, if the account balance falls below 0), or if a nontransactional operation (such as reading from or writing to a file) fails. The transaction is also aborted if the page times out before the transaction is completed.

Writing Transaction Events

A script itself cannot determine whether a transaction has succeeded or failed. However, you can write events that are called when the transaction commits or aborts. For example, suppose you have a script that credits a bank account, and you want to return different pages to the user depending on the status of the transaction. You can use the **OnTransactionCommit** and **OnTransactionAbort** events to write different responses to the user.

```
<%@ TRANSACTION=Required %>

<%
  'Buffer output so that different pages can be displayed.
  Response.Buffer = True
%>

<HTML>
  <BODY>
  <H1>Welcome to the online banking service</H1>


  <%
    Set BankAction = Server.CreateObject("MyExample.BankComponent")
    BankAction.Deposit(Request("AcctNum"))
  %>

  <P>Thank you.  Your transaction is being processed.</P>
  </BODY>
</HTML>

<%
  'Display this page if the transaction succeeds.
  Sub OnTransactionCommit()
%>
  <HTML>
    <BODY>

    Thank you.  Your account has been credited.

    </BODY>
  </HTML>

<%
  Response.Flush()
  End Sub
%>

<%
  'Display this page if the transaction fails.
  Sub OnTransactionAbort()
    Response.Clear()
%>
  <HTML>
    <BODY>

    We are unable to complete your transaction.
    </BODY>
  </HTML>
<%
    Response.Flush()
  End Sub
%>
```

Registering a Component in Component Services Manager

To participate in a transaction, a component must be registered in a COM+ application and must be configured to require a transaction. For example, if your script processes orders by calling a component that updates an inventory database and a component that updates a payment database, you would want both components to run under a transaction context. Component Services ensures that if either component fails, the entire order is rolled back and neither database is updated. Some components do not require transactions; for example, the Ad Rotator component has no need of transactions.

You use Component Services Manager to register and configure a transactional component. Components must be registered in an COM+ application. Do not put your components in the IIS in-process COM+ application; instead, create your own COM+ application. Generally, you should put all your components in one Library application. Components in Library applications can be used by multiple ASP applications and are run in the ASP application process.

You can also register transactional components in a Server application, a COM+ application that always runs in a separate process on the server. You use Server applications for your transactional components if you want to use role-based security or if you want your components to be accessible from applications on remote computers.

You must have Component Services installed to use Component Services Manager.

Object Scope

Generally, you should not store objects created from an COM component in the ASP **Application** or **Session** objects. COM objects are deactivated when the transaction is completed. Because the **Session** and **Application** objects are intended for object instances that can be used across multiple ASP pages, you should not use them to hold objects that will be released at the end of a transaction.

An ASP script is the root, or start, of a declared transaction. Any COM object used on a transactional ASP page is considered part of the transaction. When the transaction is completed, COM objects used on the page are deactivated, including objects stored in the **Session** or **Application** object. Subsequent attempts to call the session-scope or application-scope object from another transactional page will fail.

Queuing Transactions

Updates to a database on a remote server could delay or abort the completion of a transaction due to network delays or failures. Because all portions of a transaction must be committed, your application might be held up waiting for the commit or abort message from the remote server or might abort a transaction because the database update could not be sent.

For updates that must be completed simultaneously, it is appropriate to abort or even delay the completion of the transaction until all participants in the transaction can commit. For example, an airline ticket-ordering application should complete both the debit to a customer's bank account and the credit to the airline's bank account simultaneously. If an update is integral to a transaction but could occur later than the other updates, you might prefer not to keep the customer waiting for the completion of the update. For example, a transaction to order an airline ticket might also send a special meal request to a food services vendor or update the customer's mileage. These activities must be completed, but could be completed later.

You can use Message Queuing to bundle an update or set of updates into a transactional message that is delivered to a remote server. Message Queuing guarantees that updates will be delivered to the remote server, even if the network is currently unavailable. Your application receives a commit message and can continue with the transaction.

Debugging ASP Scripts

Regardless of your level of experience, you will encounter programmatic errors, or *bugs*, that will prevent your server-side scripts from working correctly. For this reason, debugging, the process of finding and correcting scripting errors, is crucial for developing successful and robust ASP applications, especially as the complexity of your application grows.

The Microsoft Script Debugger Tool

The Microsoft® Script Debugger is a powerful debugging tool that can help you quickly locate bugs and interactively test your server-side scripts. With Script Debugger, which also works with Windows Internet Explorer version 3.0 or later, you can:

- Run your server-side scripts one line at a time.
- Open a command window to monitor the value of variables, properties, or array elements, during the execution of your server-side scripts.
- Set pauses to suspend execution of your server-side scripts (using either the debugger or a script command) at a particular line of script.
- Trace procedures while running your server-side script.

Note You can use the debugger to view scripts and locate bugs, but not to directly edit your scripts. To fix bugs, you must edit your script with an editing program, save your changes, and run the script again.

Enabling Debugging

Before you can begin debugging your server-side scripts, you must first configure your Web server to support ASP debugging. For instructions and information, see Enabling ASP Debugging.

After enabling Web server debugging, you can use either of the following methods to debug your scripts:

- Manually open Script Debugger to debug your ASP server-side scripts.
- Use Internet Explorer to request an .asp file. If the file contains a bug or an intentional statement to halt execution, Script Debugger will automatically start, display your script, and indicate the source of the error.

Scripting Errors

While debugging your server-side scripts you might encounter several types of errors. Some of these errors can cause your scripts to execute incorrectly, halt the execution of your program, or return incorrect results.

Syntax Errors

A *syntax* error is a commonly encountered error that results from incorrect scripting syntax. For example, a misspelled command or an incorrect number of arguments passed to a function generates an error. Syntax errors can prevent your script from running.

Run-Time Errors

Run-time errors occur after your script commences execution and result from scripting instructions that attempt to perform impossible actions. For example, the following script contains a function that divides a variable by zero (an illegal mathematical operation) and generates a run-time error:

```
<SCRIPT LANGUAGE=VBScript RUNAT=SERVER>

  Result = Findanswer(15)

  Document.Write ("The answer is " &Result)
  Function Findanswer(x)

  'This statement generates a run-time error.

   Findanswer = x/0

  End Function

</SCRIPT>
```

Bugs that result in run-time errors must be corrected for your script to execute without interruption.

Logical Errors

A *logical* error can be the most difficult bug to detect. With logical errors, which are caused by typing mistakes or flaws in programmatic logic, your script runs successfully, but yields incorrect results. For example, a server-side script intended to sort a list of values may return an inaccurate ordering if the script contains a > (greater than) sign for comparing values, when it should have used a < (less than) sign.

Error Debugging Techniques

You can use several different debugging techniques to locate the source of bugs and to test your applications.

Just-In-Time (JIT) Debugging

When a run-time error interrupts execution of your server-side script, the Microsoft Script Debugger automatically starts, displays the .asp file with a statement pointer pointing to the line that caused the error, and generates an error message. With this type of debugging, called *Just-In-Time* (JIT) debugging, your computer suspends further execution of the program. You must correct the errors with an editing program and save your changes before you can resume running the script.

Breakpoint Debugging

When an error occurs and you cannot easily locate the source of the error, it is sometimes useful to preset a *breakpoint*. A breakpoint suspends execution at a specific line in your script. You can set one or many different breakpoints before a suspect line of script and then use the debugger to inspect the values of variables or properties set in the script. After you correct the error, you can clear your breakpoints so that your script can run uninterrupted.

To set a breakpoint, open your script with Script Debugger, select a line of script where you want to interrupt execution, and from the **Debug** menu choose **Toggle Breakpoint**. Then use your Web browser to request the script again. After executing the lines of script up to the breakpoint, your computer starts the Script Debugger, which displays the script with a statement pointer pointing to the line where you set the breakpoint.

The Break at Next Statement

In certain cases, you may want to enable the Script Debugger **Break at Next Statement** if the next statement that runs is not in the .asp file that you are working with. For example, if you set **Break at Next Statement** in an .asp file residing in an application called Sales, the debugger will start when you run a script in any file in the Sales application, or in any application for which debugging has been enabled. For this reason, when you set **Break at Next Statement**, you need to be aware that whatever script statement runs next will start the debugger.

VBScript Stop Statement Debugging

You can also add breakpoints to your server-side scripts written in VBScript by inserting a **Stop** statement at a location before a questionable section of server-side script. For example, the following server-side script contains a **Stop** statement that suspends execution before the script calls a custom function:

```
<%
  intDay = Day(Now())
  lngAccount = Request.Form("AccountNumber")
  dtmExpires = Request.Form("ExpirationDate")

  strCustomerID  =  "RETAIL" & intDay & lngAccount & dtmExpires
  'Set breakpoint here.
  Stop

  'Call registration component.
  RegisterUser(strCustomerID)
%>
```

When you request this script, the debugger starts and automatically displays the .asp file with the statement pointer indicating the location of the **Stop** statement. At this point you could choose to inspect the values assigned to variables before passing those variables to the component.

Important Remember to remove **Stop** statements from production .asp files.

JScript Debugger Statement Debugging

To add breakpoints to your server-side scripts written in JScript, insert a **debugger** statement before a suspect line of script. For example, the following script includes a **debugger** statement that interrupts execution and automatically starts Script Debugger each time the script loops through a new value.

```
<%@ LANGUAGE=JScript %>
<%
  for (var count = 1; count <= 10; count++)
  {
    var eventest = count%2
    //Set breakpoint so that user can step through execution of script.
    debugger
    if (eventest == 0)
          Response.Write("Even value is " + count + "<br>")
   }
%>
```

Remember to remove **debugger** statements from production .asp files.

Note Do not confuse the **debugger** statement with the JScript **break** statement. The **break** statement exits a currently running loop during execution and does not activate the Microsoft Script Debugger, nor pause execution.

Tips for Debugging Scripts

Aside from Script Debugger, a good set of debugging tips can greatly reduce the amount of time you spend investigating the source of scripting errors. Although most bugs result from obvious sources, misspelled commands or missing variables, certain types of logical and execution errors can be hard to find.

For more information about Microsoft Script Debugger, see **Windows Script Technologies**.

Built-in ASP Objects

Active Server Pages provides built-in objects that make it easier for you to gather information sent with a browser request, to respond to the browser, and to store information about a particular user, such as user-selected preferences. This topic briefly describes each object.

Application Object

You use the **Application** object to share information among all users of a given application.

Request Object

You use the **Request** object to gain access to any information that is passed with an HTTP request. This includes parameters passed from an HTML form using either the POST method or the GET method, cookies, and client certificates. The **Request** object also gives you access to binary data sent to the server, such as file uploads.

Response Object

You use the **Response** object to control the information you send to a user. This includes sending information directly to the browser, redirecting the browser to another URL, or setting cookie values.

Server Object

The **Server** object provides access to methods and properties on the server. The most frequently used method is the one that creates an instance of an COM component (**Server.CreateObject**). Other methods apply URL or HTML encoding to strings, map virtual paths to physical paths, and set the timeout period for a script.

Session Object

You use the **Session** object to store information needed for a particular user session. Variables stored in the **Session** object are not discarded when the user jumps between pages in the application; instead, these variables persist for the entire time the user is accessing pages in your application. You can also use **Session** methods to explicitly end a session and set the timeout period for an idle session.

ObjectContext Object

You use the **ObjectContext** object to either commit or abort a transaction initiated by an ASP script.

ASPError Object

You can use the **ASPError** object to trap ASP error and return more informative descriptions to users.

Building on the Client/Server Architecture

Before delving into the details of building a Web-based application, it might be helpful to review the architectural model of the Web, and the roles of the browser and server in that model.

Typically, cooperating applications can be categorized as either a client or a server. The client application requests services and data from the server, and the server application responds to client requests. Early two-tier (client/server) applications were developed to access large databases, and incorporated the rules used to manipulate the data with the user interface into the client application. The server's task was simply to process as many requests for data storage and retrieval as possible.

Two-tier applications perform many of the functions of stand-alone systems: They present a user interface, gather and process user input, perform the requested processing, and report the status of the request. This sequence of commands can be repeated as many times as necessary. Because servers provide only access to the data, the client uses its local resources to perform most of the processing. The client application must contain information about where the data resides and how it is organized in the database. Once the data has been retrieved, the client is responsible for formatting and displaying it to the user.

One major advantage of the client/server model was that by allowing multiple users to simultaneously access the same application data, updates from one computer were instantly made available to all computers that had access to the server. However, as the number of clients increased, the server would quickly become overwhelmed with client requests. Also, because much of the processing logic was tied to a monolithic suite of applications, changes in business rules led to expensive and time-consuming alterations to source code. Although the ease and flexibility of two-tier products continue to drive many small-scale business applications, the need for faster data access and more rapid developmental timelines has persuaded systems developers to seek out a new way of creating distributed applications.