Your Web browser is a client in the client/server environment of the Web. Up to this point, this book has focused on client-side JavaScript development in Web browsers. For you to develop a full complement of Web development skills, you also need to understand the server side of the Web—in particular, how server-side scripting fits into Web development. In this chapter, you will study server-side scripting with ASP.NET. But before getting into server-side scripting, it’s helpful to set the stage by reviewing the fundamentals of client/server architecture.
CLIENT/ SERVER ARCHITECTURE

There are many definitions of the term’s client and server. In traditional client/server architecture, the server is usually some sort of database from which a client requests information. A server fulfills a request for information by managing the request or serving the requested information to the client—hence the term, client/server. A system consisting of a client and a server is known as a **two-tier system**.

One of the primary roles of the client, or front end, in a two-tier system is the presentation of an interface to the user. The user interface gathers information from the user, submits it to a server, or back end, then receives, formats, and presents the results returned from the server. The main responsibility of a server is usually data storage and management. Heavy processing, such as calculations, on client/server systems usually takes place on the server. As desktop computers become increasingly powerful, however, many client/server systems place at least some of the processing responsibilities on the client. In a typical client/server system, a client computer may contain a front end that is used for requesting information from a database on a server. The server locates records that meet the client request, performs some sort of processing, such as calculations on the data, then returns the information to the client. The client computer may also perform some processing such as building the queries that are sent to the server or formatting and presenting the returned data. Figure 11-1 illustrates the design of a two-tier client/server system.

The Web is built on a two-tier client/server system, in which a Web browser (the client) requests documents from a Web server. The Web browser is the client user interface. You can think of the Web server as a database of Web pages. Once a Web server returns the requested document, the Web browser (as the client user interface) is responsible for formatting and presenting the document to the user.

Once you start adding databases and other types of applications to a Web server, the client/server system evolves into what is known as a three-tier client architecture. A **three-tier, or multitier, client/server system** consists of three distinct pieces: the client tier, the processing tier, and the data storage tier. The client tier, or user interface tier, is still the Web browser. However, the database portion of the two-tier client/server system
is split into a processing tier and the data storage tier. The processing tier, or middle tier, is sometimes called the processing bridge. It handles the interaction between the Web browser client and the data storage tier. Essentially, the client tier makes a request of a database on a Web server. The processing tier performs any necessary processing or calculations based on the request from the client tier, and then reads information from or writes information to the data storage tier. The processing tier also handles the return of any information to the client tier. Note that the processing tier is not the place processing takes place. The Web browser (client tier) still renders Web page documents (which requires processing), and the database or application in the data storage tier may also perform some processing. Client-side JavaScript exists at the client tier, while server-side JavaScript exists at the processing tier. Figure 11-2 illustrates the design of a three-tier client/server system.

Two-tier client/server architecture is a physical arrangement in which the client and server are two separate computers. Three-tier client/server architecture is more conceptual than physical, because the processing tier and data storage tier can be located on the same server.

Multitier client/server architecture is also referred to as \( n \)-tier architecture.

Figure 11-2  The design of a three-tier client/server system

An important question in the design of any client/server system is deciding how much processing to place on the client. In the context of Web site development with JavaScript, you must decide whether to use client-side or server-side JavaScript. This decision may sound confusing, because client-side JavaScript and server-side JavaScript share much of the same core language. However, whether to use client-side or server-side JavaScript is important consideration because the choice you make can greatly affect the performance
of your program. A general rule of thumb is to allow the client to handle the user inter-
face processing and light processing, such as data validation, but have the server perform
intensive calculations and data storage. This division of labor is especially important when
dealing with the Web. Unlike with clients on a private network, it’s not possible to know in
advance the computing capabilities of each client on the Web. You cannot assume that
each client (browser) that accesses your client/server application (Web site) has the nec-
essary power to perform the processing required by the application. For this reason, inten-
sive processing should be placed on the server.

Because servers are usually much more powerful than client computers, your first instinct
may be to let the server handle all processing and only use the client to display a user
interface. Although you do not want to overwhelm clients with processing they cannot
handle, it is important to perform as much processing as possible on the client for sev-
eral reasons:

- Distributing processing among multiple clients creates applications that are
  more powerful because the processing power is not limited to the capabilities
  of a single computer. Client computers become more powerful every day, and
  advanced capabilities such as JavaScript and DHTML are now available in
  local Web browsers. Thus, it makes sense to use a Web application to har-
  ness some of this power and capability. (A Web application is a program that
  executes on a server but that clients access through a Web page loaded in a
  browser.)

- Local processing on client computers minimizes transfer times across the
  Internet and creates faster applications. If a client had to wait for all process-
  ing to be performed on the server, a Web application could be painfully slow
  over a busy Internet connection.

- Performing processing on client computers lightens the processing load on
  the server. If all processing in a three-tier client/server system is on the
  server, the server for a popular Web site could become overwhelmed trying
to process requests from numerous clients.

The term distributed application is used to describe multiple computers sharing
the computing responsibility for a single application.
Chapter 5 touched on three-tier client/server processing in relation to server-side scripting languages that execute on a Web server. Some of the more popular server-side scripting languages that are used to process form data include Common Gateway Interface (CGI), Java Server Pages (JSP), and Active Server Pages (ASP). Server-side scripting languages exist on the processing tier and have the ability to handle communication between the client tier and the data storage tier. At the processing tier, a scripting language usually prepares and processes the data in some way before submitting it to the data storage tier. Additionally, server-side JavaScript can interact closely with client-side JavaScript, because they share the same basic programming features. For these reasons, many developers prefer server-side scripting languages for handling the processing tier of a three-tier client/server system.

Server-side scripting languages are not without their drawbacks. One of their biggest disadvantages is that server-side scripting languages are proprietary and vendor specific. There is no server-side scripting language standard similar to ECMAScript. Although server-side scripting languages are proprietary and vendor specific, most implementations use much of the same core language syntax that you have learned for client-side JavaScript. You can write server-side scripts using many of the skills you have already learned in client-side JavaScript, regardless of the server-side scripting language you use.

In this chapter, you will use ASP.NET, which is a server-side scripting language created by Microsoft. ASP.NET is the latest version of Active Server Pages and is part of the Microsoft Visual Studio .NET line of software development products. You can create ASP.NET using one of the following Windows platforms:

- Windows 2000 (Professional, Server, and Advanced Server)
- Windows XP Professional
- Windows Server 2003 family for both client and server applications

To use ASP.NET, you must also install Internet Information Services (IIS), which are used for creating, configuring, and managing Web sites. IIS is available with each of the preceding platforms. See your Windows documentation for information on installing and configuring IIS.

The examples and figures in this chapter were created using Windows XP Professional.

The Microsoft Visual Studio .NET line of software development products includes a platform called the .NET Framework, which is designed for creating, deploying, and running Web applications and services. In order to create ASP.NET applications, you must install the .NET Framework. ASP.NET is automatically installed as part of the

Because the goal of this book is to teach JavaScript, this chapter provides only a brief overview of ASP.NET. The installation and configuration requirements for ASP.NET, the .NET Framework, or IIS are beyond the scope of this book. For installation and configuration procedures, along with additional information on working with ASP.NET, see the Microsoft ASP.NET page at http://msdn.microsoft.com/asp.net/. If you are in a classroom setting, ask your instructor for specific instructions on how to work with ASP.NET at your institution.

The ASP.NET techniques you study in this chapter are identical in both ASP and ASP.NET. Although the chapter focuses on ASP.NET, you can also perform the exercises in this chapter with ASP if you do not have access to ASP.NET. Active Server Pages are supported by Internet Information Server 3.0 and higher running on Windows NT Server 4.0 and Windows 2000, Peer Web Services running on Windows NT Workstation 4.0, and Personal Web Server running on Windows 95/98. Older versions of ASP (without the .NET extension) use a file extension of .asp.

You save ASP.NET documents in a folder that you designate as a virtual directory in IIS. Once you create a virtual directory in IIS, you can open any files it contains in a Web browser using the following syntax: http://computer_name/directory/file. For example, if your computer is named personal_computer and you create a virtual directory named interests, you can open a document named index.html by typing the following statement in a Web browser's address bar: http://personal_computer/interests/index.html. Alternatively, you can replace the computer name with localhost, as follows: http://localhost/interests/index.html. The term localhost is an alias for the name of a computer.

You can change the name of your computer using the System Properties dialog box in the Control Panel.

In this chapter, you will create a Web site for a company named WebAdventure; the Web site will use ASP.NET to store the number of hits it receives, and it will include an area where visitors can sign a guest book. The program you create will not be a complete, robust Web site, because it includes only a hit counter and a database. However, it will serve to demonstrate how to create a basic program with ASP.NET.

Next, you will create a virtual directory in IIS for the WebAdventue Web site. The following instructions are for Windows 2000, Windows XP, and Windows Server 2003. If
you are running Peer Web Services or Personal Web Server, refer to your Windows documentation for information on creating a virtual directory.

To create a virtual directory in IIS for the WebAdventure Web site:

1. Open the Control Panel from the Start menu.

2. If you are using Windows XP, click the **Switch to Classic View** link to display the Control Panel icons.

3. Use the **Administrative Tools** icon to open the Administrative Tools window.

4. Select **Internet Information Services**. The Internet Information Services window opens.

5. Click the plus sign next to the icon that represents your computer, and then click the plus sign next to the Web Sites folder, if necessary.

6. Click the plus sign next to the Default Web Site icon that appears beneath the computer name icon. Figure 11-3 shows the Internet Information Services window on a computer named DONGOSSELIN.

7. Verify that the Default Web Site icon is selected, click **Action** on the menu bar, point to **New**, and then click **Virtual Directory**. The Virtual Directory Creation Wizard appears.

8. In the introductory dialog box of the Virtual Directory Creation Wizard, click the **Next** button to display the Virtual Directory Alias dialog box.

9. In the Virtual Directory Alias dialog box, type `webadventure`, and then click the **Next** button. The Web Site Content Directory dialog box appears.
10. Type the path where you store your JavaScript projects. For example, if your JavaScript projects are stored in a folder named JavaScript_Projects on your C: drive, you would type C:\JavaScript_Projects. Click the Next button when you are finished. The Access Permissions dialog box appears.

11. Leave the options in the Access Permissions dialog box set to their default values, and click the Next button to display the final Virtual Direction Wizard Creation dialog box.

12. Click Finish to create the virtual directory.

13. After you create the webadventure virtual directory, close Internet Information Services and the Administrative Tools window.

**Creating ASP.NET Applications**

ASP.NET scripts are created as text files, the same as XHTML documents, and can contain both client-side JavaScript code and server-side JavaScript code. Documents for ASP.NET applications have the .aspx file extension. ASP applications files have an .asp extension. When a client requests an ASP.NET document, the Web server executes any server-side scripting code before serving the document to the client. Once the client Web browser receives the document, it executes the client-side JavaScript.

Be aware that the first time a client requests an ASP.NET document from a server, the server compiles the document, regardless of whether it contains server-side JavaScript code. The term **compiling** refers to the processing and assembly of programming code into an executable format. The difference between interpreting and compiling is that, while interpreted programs (such as JavaScript programs) are processed and assembled into an executable format each time they execute, compiled programs only need to be recompiled when their code changes. You do not need to compile a server-side script yourself because ASP.NET automatically recognizes any changes to an ASP.NET application and recompiles the application the next time a client requests it. Thus, installing an ASP.NET application is as simple as placing the documents that make up the application in one of the server’s directories. Additionally, an ASP.NET application starts automatically the first time a Web server receives a request for one of the application’s pages. The term **ASP.NET application** refers to a collection of related ASP.NET documents that exist in the same root directory.

Because the compilation process requires extra processing time, you should not use the .aspx or .asp extensions with XHTML documents that do not contain server-side JavaScript code, because XHTML documents do not need to be compiled.
Code Declaration Blocks

You define ASP.NET global variables and functions within code declaration blocks, which you create with `<script>` elements, the same as client-side JavaScript. You use three attributes in a code declaration block: `runat="server"`, `language`, and `src`. The `runat="server"` attribute is required in order to identify the script section as an ASP.NET script. The `language` attribute identifies the scripting language used within the code declaration block. You can use three scripting languages with ASP.NET: VB (Visual Basic), JScript (JavaScript), and C# (a .NET Framework programming language that is part of Visual Studio). Although the `language` attribute is optional, ASP.NET uses Visual Basic by default, so you must include the `language` attribute in order to use JavaScript with your ASP.NET applications. The `src` attribute is the same attribute that is used in client-side JavaScript to identify a source document containing scripting code.

A major difference between using the `<script>` element for ASP.NET instead of client-side JavaScript is that all code, with the exception of global variables, must be contained within a function. For example, the following code contains an example of a code declaration block that includes a single `Write()` method of the `Response` object within a function named `sayHello()`.

```html
<script language="JScript" runat="server">
function sayHello() {
    Response.Write("Hello World");
}
</script>
```

By comparison, the following code declaration script would generate a compile error because the statement is not contained within a function:

```html
<script language="JScript" runat="server">
Response.Write("Hello World");
</script>
```

Because XHTML does not recognize the ASP.NET `run` attribute and because the `language` attribute is deprecated, you cannot validate the ASP.NET documents you create. However, even though you cannot validate your ASP.NET documents, you should still write well-formed XHTML elements and attributes, especially for any elements and attributes that will be returned as a response to the user.

Notice that the `Write()` method in the preceding examples is appended to the `Response` object in the same manner that the `write()` method in client-side
JavaScript is appended to the `Document` object. Also notice that the `Response` object and `Write()` method are written with their first letter in uppercase. In ASP.NET, objects and methods are usually written with uppercase first letters. You will learn about the `Response` object later in this section. The `Write()` method works the same as the client-side `write()` method, except that instead of designating text to be output to the browser, it designates information to be returned to the client.

You cannot execute a function contained within a code declaration block. Instead, you must execute a function from a code render block, which you will study next.

Next, you will start creating a document that will be used as the start page for the WebAdventure Web site.

To start creating a document that will be used as the start page for the WebAdventure Web site:

1. Open your text editor and create a new document.
2. Type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “WebAdventure” as the content of the `<title>` element. Your document should appear as follows:

   ```html
   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
   <html>
   <head>
   <title>WebAdventure</title>
   </head>
   <body>
   </body>
   </html>
   ```
3. Add the following ASP.NET code declaration block to the document head. The script section uses the `Date` object to generate the date, which it sends to the client using a `Response.Write()` statement.

   ```javascript
   function timeStamps() {
     var now = new Date();
     var day = now.getDay();
     var date = now.getDate();
     var year = now.getFullYear();
     var month = now.getMonth();
     var hours = now.getHours();
     var minutes = now.getMinutes();
     var seconds = now.getSeconds();
     var days = new Array();
     days[0] = "Sunday"; days[1] = "Monday";
   }
   ```
days[6]="Saturday";
var display = days[day] + " " + month + "/" + date + "/" +
+ year + " " + hours + ":" + minutes + ":" + seconds;
Response.Write("The current date and time are ")
Response.Write(display);
Response.Write("<hr />");

4. Add to the document body the following heading element, a form that con-
tains a text field for the user's first name, and a Submit button that calls the
main document, named HomePage.aspx. The Submit button submits the
value in the text field to the HomePage.aspx document.

<h3>Please enter your first name and click Continue
to proceed to our home page.</h3>
<form method="post" action="HomePage.aspx">
<p>First Name: <input type="text" name="first" />
<input type="submit" value="Continue " /></p>
</form>

5. Save the document as StartPage.aspx in your Chapter folder for Chapter 11.

**Code Render Blocks**

ASP.NET also uses the script delimiters `<%` and `%>` to designate server-side JavaScript
code. A **delimiter** is a character or a sequence of characters used to mark the beginning
and end of a code segment. You include within the script delimiters any commands that
are valid for the scripting language you are using.

```html
<% Response.Write("Hello World"); %>
```

The preceding statement creates a **code render block**, which executes ASP.NET code
within a Web page document. You can declare the `language` attribute in a code ren-
der block, but you must use an **ASP processing directive**, which provides a Web server
with information on how to process the scripts in an ASP document. ASP processing
directives are created using the `<%@` ... `%>` delimiters. You can only declare one pro-
cessing directive on a page, so it is usually declared at the top of a document or above
the first code render block. For example, the following statement includes an ASP pro-
cessing directive, which declares that the code render block is written in JScript:

```html
<%@ language="JScript" %>
<% Response.Write("Hello World"); %>
```
One of the primary reasons to use a code render block is to include ASP.NET inside an XHTML element. As an example, consider the `Session` object, which is used in ASP.NET to create custom client properties. Assume that you have created a custom property, named `email`, in the `Session` object, and you want to use that property in a link that uses the XHTML `mailto:` protocol. To return the custom `email` property of the ASP.NET `Session` object and use it in the link, you use syntax similar to the following:

```
<a href="mailto:<%= Response.Write(Session.Contents('email')) %>">
  Click here to send an e-mail</a>
```

You cannot declare a function within a code render block.

Notice in the preceding code that you refer to the `email` property of the `Session` object as an argument of the `Contents` object. Properties in ASP.NET are stored in data structures known as `collections` instead of as properties of objects. The `Contents` object referenced in the preceding code is actually called the `Contents` collection. You will learn about collections at the end of this section.

An important difference between code declaration blocks and code render blocks is that any variables you define within a code render block are only available to other code render blocks on the page. You can include multiple code declaration blocks and code render blocks in a document. However, variables that you want to be globally available to all other ASP.NET code sections must be declared within a code declaration block.

Although you can include multiple code declaration blocks and code render blocks in a document, you can only use one scripting language: VB (Visual Basic), JScript (JavaScript), or C#. In other words, the value you assign to the language attributes in multiple script sections must match.

The **output directive** sends the result of an expression to a user’s Web browser (the client). The syntax for the output directive is `<%= expression %>`. For example, if your script includes a variable named `salesTotal` that has been assigned a value of $24.95, then the output directive `<%= salesTotal %>` sends the value $24.95 to the client Web browser.

The output directive is equivalent to the `Response` object `Write()` method.
The following code shows how to use the output directive in a code render block. The example combines the output directive with the `Response` object `Write()` method, and includes the `language` attribute to designate JavaScript as the default scripting language. The expression in the output directive calculates `firstNum` and `secondNum`, then sends the text “The result of 10 minus 2 is 8” to the user's Web browser. Figure 11-4 shows the output.

```html
<html>
<head>
<title>Code Render Block</title>
</head>
<body>
<p>
<%@language="JScript" %>
<% var firstNum = 10, secondNum = 2;
Response.Write("The result of " + firstNum + " minus " + secondNum + " is "); %>
<%= firstNum - secondNum %>
</p>
</body>
</html>
```

Figure 11-4  ASP.NET code render block example
Next, you add a processing directive and a code render block to the StartPage.aspx document.

To add a processing directive and a code render block to the StartPage.aspx document:

1. Return to the **StartPage.aspx** document in your text editor.

2. Add the following processing directive to the top of the document. The processing directive declares JScript as the script language.
   
   `<%@ language="JScript" %>`

3. Now add a code render block after the closing `<script>` tag in the document head. The code render block executes the `timeStamp()` function you added in the last exercise.
   
   `<% timeStamp() %>`

4. Save the **StartPage.aspx** document, and then open it in your Web browser by typing the following URL in the address box: `http://localhost/webadventure/Chapter.11/Chapter/StartPage.aspx`. You cannot open StartPage.aspx as a local file using the Open command from your Web browser’s File menu. You must type the full URL to your server.

5. After the document opens, view the source document. (In Internet Explorer, select Source from the View menu.) In the text editor that opens, notice that the document does not contain the ASP.NET statements you added. Instead, only the output from the `Response.Write()` statements is available to the client. Figure 11-5 shows how the document appears in a Web browser.


![Figure 11-5](StartPage.aspx)
Mixing XHTML and ASP.NET

The ASP.NET code within code declaration blocks and code render blocks execute on a Web server before the page is sent to a user. If users were to view the source document after they received the ASP document, they would not see any `<script>` elements, `<%...%>` script delimiters, or ASP.NET code they contain. Instead, the client shows only the results returned by the code. For example, the following code shows an ASP document containing client-side JavaScript as well as ASP.NET code render blocks and code declaration blocks. Notice that the function declared within the code declaration block is executed in the last code render block.

```html
<html>
<head>
<title>Mixed Example</title>
<script type="text/javascript">
<!-- HIDE FROM INCOMPATIBLE BROWSERS
document.write("<h2>Client-side JavaScript</h2>" );
document.write("<p>This line is generated by client-side JavaScript.</p>" );
// STOP HIDING FROM INCOMPATIBLE BROWSERS -->
</script>
</head>
<body>
<%@ language="JScript" %>
%</body>
</html>
```

The following code shows how the preceding document appears once a client receives it. Notice that the client-side JavaScript code is the same, but the ASP.NET code has been converted to elements and text. Figure 11-6 shows how the text and elements appear in a Web browser.

```html
<html>
<head>
<title>Mixed Example</title>
<!-- HIDE FROM INCOMPATIBLE BROWSERS
</head>
<body>
<% codeBlockText(); %>
</body>
</html>
```
You can see in the preceding example that XHTML elements and text can be interspersed with script delimiters. However, you can go even further and include XHTML elements and text as part of an ASP.NET decision-making structure, such as an if...else statement. For example, in the following code, because each if or else...if statement is enclosed by script delimiters, it is unnecessary to use the Write() method of the Response object to output the correct greeting. Only one greeting is returned, based on the results of the if...else statement.

```c#<% if (Request.Form("language") == "Spanish") %>
<p>Buenos Dias</p>
<% else if (Request.Form("language") == "German") %>
<p>Guten Tag</p>```
Creating ASP.NET Applications

You could also write the code within a single pair of script delimiters, as the following code illustrates. However, the greetings are not treated as XHTML elements or text, because they are enclosed by the script delimiters. For this reason, each greeting must include a `Response.Write()` statement to be returned to the client.

```csharp
<% if (Request.Form("language") == "Spanish")
    Response.Write("<p>Buenos Dias</p>");
else if (Request.Form("language") == "German")
    Response.Write("<p>Guten Tag</p>");
else if (Request.Form("language") == "Italian")
    Response.Write("<p>Buon Giorno</p>");
else if (Request.Form("language") == "French")
    Response.Write("<p>Bonjour</p>");
else
    Response.Write("I don’t speak your language!");
%>
```

The following code shows another example of interspersing ASP.NET with XHTML elements and text to generate a customized Web page. The example shows an XHTML document that could be generated in response to a client request for information. Assume that the document from which the client request originates includes a form field named `clientName`. The example uses the `clientName` property of the `Content` collection of the `Request` object and uses an output directive to intersperse the property throughout the text to be returned to the client, creating a customized response. The example also assumes that the company’s e-mail address is stored in the `email` property of the `Content` collection of the `Application` object. Figure 11-7 shows the output in a Web browser, assuming that “Don” is the value of the `clientName` property.

```html
<html>
<head>
    <title>Mixed Example</title>
</head>
<body>
<p>Dear <%= Request.Form("clientName") %>:</p>
<p>Thank you for your interest in our Web development services. As you know, it is difficult to keep up with rapidly changing technology. With WebAdventure’s help, you can be confident that your Web site will be on the cutting edge. Remember,
```
<%= Request.Form("clientName") %>, WebAdventure is here for you. For more information, click <a href="mailto:<%= Application.Contents("email") %>">here</a> to send us an e-mail.</p></body></html>

OBJECT COLLECTIONS

Before learning about ASP.NET core objects, you need to understand collections. Collections are data structures similar to arrays that store variables in ASP.NET core objects. The syntax for assigning a variable to a collection is `object.collection("variable") = value;`. You then refer to the variable in ASP.NET code by using the syntax `object.collection("variable");`. For instance, the Application object and Session object both contain separate Contents collections, which store custom variables. The following code creates a name variable in the Session object Contents collection, assigns it a value of “Don Gosselin”, and then sends the value to the client browser, using a Response.Write() statement:

```jscript
<%@ Language="JScript" %>
<% Session.Contents("name") = "Don Gosselin"; Response.Write("Your name is " + Session.Contents("name")); %>
```

Collections are commonly used in Visual Basic programming. Much of ASP.NET non-JavaScript syntax comes from Visual Basic programming, because the ASP.NET default scripting language is VBScript, which is derived from Visual Basic.
If a variable name you assign to a collection is unique throughout all of the collections of an object, then you can eliminate the collection name when referencing the object in code. For example, if the `Session` object contains only one `name` variable in all of its collections, then you can refer to the `name` variable in code by using a statement such as `Response.Write(Session("name"));`. It is usually safer, however, to include the collection name in order to eliminate any uncertainty as to what collection contains a specific variable. Additionally, eliminating the collection name could cause bugs in your program if you later add the same variable name to another collection in the same object. Therefore, the code in this chapter includes collection names when referencing variables.

Each variable in a collection is numbered, similar to the way element numbers are assigned to arrays. You can refer to collection variables by using numbers instead of variable names. For example, the following code creates three variables, `firstName`, `lastName`, and `email`, in the `Contents` collection of the `Session` object, then returns the values to the client, using the collection number of each variable.

```<%@ Language="JScript" %>
<%
  Session.Contents("firstName") = "Don";
  Session.Contents("lastName") = "Gosselin";
  Session.Contents("email") = "dongosselin@compuserve.com";
  Response.Write(Session.Contents(0));
  Response.Write(Session.Contents(1));
  Response.Write(Session.Contents(2));
%>
```

Be aware that the number assigned to a variable in a collection can change if you remove items from the collection. You should not assume that a number always represents the same variable in a collection.

You can remove items from a collection using the `Remove()` and `RemoveAll()` methods of the `Contents` collection. Both the `Application` object and `Session` object contain `Contents` collections.

ASP.NET collections support a `count` property, which returns the number of variables in a collection. You can use the `count` property in a looping statement to cycle through the variables in a collection. For example, the following code uses a `for` loop to return the variables in the `Contents` collection to the client. The `for` loop conditional evaluation compares the `curVariable` variable to the `count` property. While `curVariable` is less than the `count` property, the `for` loop continues iterating through the collection.
ASP.NET Core Objects

ASP.NET does not recognize a Web browser’s Document and Window objects, as client-side JavaScript does. Instead, ASP.NET recognizes five built-in objects, Request, Response, Session, Application, and Server objects, that function at the processing tier between the client tier and the data storage tier (if there is one). The Request, Response, Session, Application, and Server objects are built into ASP.NET. However, they each have different lifetimes and availability. You use each of the objects to access specific types of information in the processing tier and for storing state information on the server. You will now study each of these objects in detail.

Request Object

The Request object represents the current URL request from a client. ASP.NET creates a new Request object each time a client requests a URL. For example, if users click a link or select a new URL in their browser, then ASP.NET creates a Request object. ASP.NET also creates a Request object when client-side JavaScript uses the document.location or history.go() methods. The Request object has the shortest lifetime of all the ASP.NET built-in objects, because it exists only until the current request is fulfilled.

The Request object contains several collections that contain information about the client request. Table 11-1 lists common collections of the Request object.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientCertificate</td>
<td>Field values in the client certificate sent with the request</td>
</tr>
<tr>
<td>Cookies</td>
<td>Cookies sent with the request</td>
</tr>
<tr>
<td>Form</td>
<td>The value of named form elements in the document displayed in the browser</td>
</tr>
<tr>
<td>QueryString</td>
<td>The name=value pairs appended to the URL in a query string</td>
</tr>
<tr>
<td>ServerVariables</td>
<td>Environment variables</td>
</tr>
</tbody>
</table>
The **Form** collection of the **Request** object contains variables representing form elements from the requesting Web page. ASP.NET takes all of the named elements in a form on the user's browser and adds them as variables to the **Form** collection of the **Request** object. Recall that when you click a form's Submit button, each field on the form is submitted to the server as a name=value pair. The name portion of the name=value pair becomes a variable name in the **Form** collection, and the value portion is assigned as the value of the variable. The following code contains a typical form:

```html
...<br />
    <input type="text" name="name" size="50" /><br />
    Address<br />
    <input type="text" name="address" size="50" /><br />
    City, State, Zip<br />
    <input type="text" name="city" size="38" />
    <input type="text" name="state" size="2" maxlength=2 />
    <input type="text" name="zip" size="5" maxlength="5" />
    E-Mail<br />
    <input type="text" name="email" size="50" />
</p>
<p><input type="reset" /> <input type="submit" />
</p>
</form>
</body>
</html>
```

Upon submitting the preceding form to an ASP.NET document named ProcessOrder.aspx, the field names and values are assigned as variables to the **Request** object **Form** collection. You refer to each form variable in the **Request** object **Form** collection by using the following statements:

```csharp
    Request.Form("name")
    Request.Form("address")
    Request.Form("city")
    Request.Form("state")
    Request.Form("zip")
    Request.Form("email")
```

When name=value pairs are attached to a URL as a query string, they are assigned as variables to the **Request** object **QueryString** collection. Consider the following code, which appends a query string to a URL:

```html
<a href="http://www.URL.com/TargetPage.aspx?firstName=Don &lastName=Gosselin&occupation=writer">Link Text</a>
```
After users click the link, TargetPage.aspx opens. Any ASP.NET code thereafter can refer to firstName, lastName, and occupation as variables in the Request object QueryString collection as follows:

```csharp
Request.QueryString("firstName")
Request.QueryString("lastName")
Request.QueryString("occupation")
```

Now that you understand the basics of the Request object, you create the WebAdventure home page document.

To create the WebAdventure home page document:

1. Create a new document in your text editor.
2. Type the processing directive, as follows:
   ```csharp
   < %@ language= "JScript" %>
   ```
3. Now type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “WebAdventure Home Page” as the content of the `<title>` element.
4. Add the following heading elements to the document body:
   ```html
   <h1>WebAdventure Home Page</h1>
   <h2>Welcome</h2>
   ```
5. Add the following paragraph and horizontal rule to the end of the document body. The paragraph uses the Request object with the output directive to insert the value of the first name text field from the StartPage.aspx document:
   ```html
   <p>Hello <%= Request.Form("first") %>! Welcome to WebAdventure, Inc. We are an industry leader in JavaScript development. For more information, send a message to <a href="mailto:information@webadventure.com">information@webadventure.com</a>.</p>
   ```
   ```html
   <hr />
   ```
7. Open StartPage.aspx in your Web browser. Enter your name in the First Name text box and click the Continue button to open HomePage.aspx. Figure 11-8 shows how the document appears in a Web browser. You can see that the name you typed in StartPage.aspx has been inserted into HomePage.aspx.
8. Close your Web browser window.
The Response object sends output and information back to the client. You have used the Response object Write() method to send text back to the client. Although this book discusses only the Write() method, the Response object includes several other methods, as well as properties that are useful in constructing a response to return to the client. For example, the Response object Redirect() method sends a client to a different Web page, in the same manner as the href property of the client-side JavaScript Location object.

Visit the Microsoft Developer Network at msdn.microsoft.com/ for a complete listing of the Response object properties and methods.

The Response object contains a Cookies collection that is used for setting cookies on a client system. The Cookies collection is the only collection contained in the Response object.

Session Object

A new Request object is instantiated each time a client requests an ASP.NET URL, and then destroyed once the URL is delivered to the client. However, an ASP.NET application may be composed of multiple documents. Because the Request object is destroyed once the URL is delivered to the client, you cannot use the same Request object with different pages in an application. If you want to preserve client information across multiple pages in an ASP.NET application, you must use the Session object. The Session object temporarily stores specific client information and makes that information available to all the pages in an ASP.NET application. A Session object is instantiated the first time a client accesses a URL in a given application.
You use the Contents collection of the Session object to store information about the user in your application. For example, you may store the values from the Request object Form collection in the Session object Contents collection in order to make the form information available across all the pages in an ASP.NET application. However, you cannot directly assign the values from one collection to another collection using statements similar to the following:

```
Session.Contents("name") = Request.Form("name");
```

If you use the preceding statement you receive a compiler error because ASP.NET attempts to assign the entire Request.Form("name") object to the name variable of the Session object’s Contents collection. In order to force ASP.NET to assign just the value of a collection variable, you must convert the variable to a string literal using ASP.NET’s String() method. The syntax for converting a collection variable to a string literal is ```String(object.collection("variable"));``` The following code shows an example of an ASP.NET application that assigns the values of form fields to variables of the Session object Contents collection. The statements use String() methods to convert each variable in the Request object Form collection to string literals.

```csharp
<%
Session.Contents("name") = String(Request.Form("name"));
Session.Contents("address") = String(Request.Form("address"));
Session.Contents("city") = String(Request.Form("city"));
Session.Contents("state") = String(Request.Form("state"));
Session.Contents("zip") = String(Request.Form("zip"));
Session.Contents("email") = String(Request.Form("email"));
%>
```

The Session object includes several properties. Table 11-2 lists common properties of the Session object.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodePage</td>
<td>The character set used for a given language</td>
</tr>
<tr>
<td>LCID</td>
<td>Identifies a user's, region's, or application's preferred human language</td>
</tr>
<tr>
<td>SessionID</td>
<td>The user's session identification</td>
</tr>
<tr>
<td>Timeout</td>
<td>The lifespan of the Session object</td>
</tr>
</tbody>
</table>

It's important to keep in mind that the ASP.NET Session object has a default lifespan of 10 minutes. A default lifespan is necessary because the Web does not contain any links between a client and a server as a real network does; it relies instead on HTTP to send requests and responses back and forth over the Internet. Therefore, ASP.NET has no way of
knowing whether a client has left the Web site. For example, when users access a Web site, they are only requesting a document. Once an ASP.NET server returns the requested document, it has no way of knowing whether the client will request other Web pages. From the server’s point of view, the only possible indication that a client hasn’t left the Web site occurs when the client requests another document. If the client fails to make another request within ten minutes, the ASP.NET Session object that was created for the client’s initial request is deleted. If you want to allocate more time before a Session object is deleted, you use the Timeout property, which determines the lifespan of a Session object. The syntax for the Timeout property is \texttt{Session.Timeout = \textit{minutes};}. If you want to increase a Session object lifespan to 20 minutes, you use the statement \texttt{Session.Timeout = 20;}.

The Contents collection of the Session object also includes three methods: Abandon(), Remove(), and RemoveAll(). The Abandon() method completely destroys a Session object. The Remove() and RemoveAll() methods remove items from Session object collections.

Next, you will add a property to the Session object for the WebAdventure script.

To add a property to the Session object for the WebAdventure script:


2. Add the following code render block to the end of the document head. The code render block contains a simple if statement that uses the logical not (!) operator to check if the sameClient variable exists in the Session object. If the sameClient variable does not exist, it is created and set to an initial value of true. The sameClient variable indicates that a client is working within the same session, regardless of whether it changes to a different page in the application. You will use this value later when creating the page hit counter.

   ```
   <%
   if (!Session.Contents("sameClient")) {
     Session.Contents("sameClient") = "true";
   }
   %>
   

\section*{Application Object}

An Application object is used for storing global application information that can be shared by all clients accessing the application. Each application has its own Application object. An ASP.NET application automatically starts the first time a client requests one of the application pages. ASP.NET applications run until the server is shut down.
You create your own variables for the Application object in its Contents collection. One common property for the Application object is some type of unique number used to identify clients. You can create a variable in the Application object Contents collection that keeps track of the last assigned number, and then use a function to update the number and assign it to a client. For example, suppose you have a Web application that takes online orders. Each time a client accesses the application, you want to assign a unique invoice number. The following code assigns to the curInvoiceNum variable the value of the lastInvoiceNum variable of the Application object Contents collection. The curInvoiceNum variable is then incremented by one. Then, a new invoiceNum variable is created in the Session object Contents collection and assigned the value of the curInvoiceNum variable. You place this code in the first page of the online order application to assign an invoice number to a client that accesses the page.

```<% curInvoiceNum = Application.Contents("lastInvoiceNum"); curInvoiceNum = ++curInvoiceNum; Application.Contents("lastInvoiceNum") = curInvoiceNum; Session.Contents("invoiceNum") = curInvoiceNum; %>`

You can remove items from an Application object collection using the Remove() and RemoveAll() methods of the Contents collection.

TIP

The variables of the ASP.NET Application object are available to all clients that access the application, and it is possible that one client may try to access a property before another client is through with it. To prevent one client from accessing a variable of the Application object until another client is through with it, you use the Application object Lock() and UnLock() methods. The Lock() method prevents other clients from accessing properties of the Application object, and the Unlock() method cancels the Lock() method. You place the Application.Lock(); statement before any code that accesses Application object properties. Following the last statement that accesses Application object properties you place the Application.UnLock(); statement. For example, to prevent data integrity problems with the lastInvoiceNum code example, you use the Lock() and UnLock() methods as follows:

```<% Application.Lock(); curInvoiceNum = Application.Contents("lastInvoiceNum"); curInvoiceNum = ++curInvoiceNum; Application.Contents("lastInvoiceNum") = curInvoiceNum; Session.Contents("invoiceNum") = curInvoiceNum; Application.UnLock(); %>`
Now that you understand how to preserve variables across multiple user sessions, you will add to the HomePage.aspx document a counter that tracks the total number of hits.

To add a hits counter to the HomePage.aspx document:

1. Return to the **HomePage.aspx** document in your text editor.

2. In the code render block you added in the last exercise, add the following statements above the `Session.Contents("sameClient") = "true";` statement. The first statement locks the application. The **if** statement uses a logical not (!) operator to check if the **counter** variable of the Application object exists. The **counter** variable will hold the number of hits the page receives. If the **counter** variable does not exist, it is created and assigned an initial value of one. If the **counter** variable does exist, its value is assigned to the **curNumber** variable, which increments it by one. The new value in the **curNumber** variable is then assigned to the **counter** variable. Finally, the application is unlocked.

   ```csharp
   Application.Lock();
   if (!Application.Contents("counter"))
       Application.Contents("counter") = 1;
   else {
       var curNumber = Application.Contents("counter");
       curNumber = ++curNumber;
       Application.Contents("counter") = curNumber;
   }
   Application.UnLock();
   
   3. Next, add the following code to the end of the document body to display the number of times the page has been accessed:

```csharp
   <p>This page has been accessed <%= Application.Contents("counter") %> times.</p><hr />
```


5. Open **StartPage.aspx** in your Web browser. Enter your name in the First Name text box and click the **Continue** button to open HomePage.aspx. The page counter should start at one. Figure 11-9 shows the page after it has been accessed twice.

The Server object provides an ASP.NET application with access to properties and methods that are accessible by all applications on the server. The ASP.NET Server object does not contain any collections, and you cannot add custom variables to it. The ASP.NET Server object is used mainly for managing the way an ASP.NET application behaves on the server.

The ASP.NET Server object includes a single property, ScriptTimeout. The ScriptTimeout property determines how long an ASP.NET application can run before the server stops it. By default, a server allows an ASP.NET application to run for 90 seconds. If you anticipate that your application will need more time to run, increase its running time, using the syntax Server.ScriptTimeout = seconds; . For example, to increase the maximum running time of an application to two minutes, use the statement Server.ScriptTimeout = 120; . Table 11-3 lists common methods of the Server object.

You will use several methods of the Server object in Chapter 12 to access databases.

NOTE
Creating a Guest Book

Next, you will add a guest book to the ASP.NET version of the WebAdventureHome program. You will save guest entries as variables in the Application object Contents collection. Saving guest entries in a core object is not the best method of creating a guest book, because the list will be lost if you need to restart your server. It’s more efficient to save the entries to a database. However, the main purpose of this exercise is to help you understand how to work with ASP.NET core objects. You will learn how to use ASP.NET to read and write to databases in Chapter 12.

First, you will add some text and a form to HomePage.aspx.

To add some text and a form to HomePage.aspx:

2. Add the following elements and text to the end of the document body. The text includes a guestCounter variable of the Application object, which returns the number of people who have signed the guest book.

   <h2>Guest Book</h2>
   <p>Please join the other <%= Application.Contents("guestCounter") %> people who have signed our guest book. We have already entered your first name for you. Please enter your last name and e-mail address, then click the <strong>I’m Finished</strong> button. If you would like to see a list of other people who have signed our guest book, click the <strong>See Who’s Been Here</strong> button.</p>

3. Next, add to the end of the document body the following form to be used for signing the guest book. The First Name text box uses the Request object to set the value of the first name text field to the value of the text

---

### Table 11-3: Common methods of the Server object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateObject()</td>
<td>Instantiates a server component</td>
</tr>
<tr>
<td>Execute()</td>
<td>Executes an ASP.NET document</td>
</tr>
<tr>
<td>GetLastError()</td>
<td>Returns an ASPError object, which describes an error condition that occurred</td>
</tr>
<tr>
<td>HtmlDecode()</td>
<td>Decodes an encoded string</td>
</tr>
<tr>
<td>HtmlEncode()</td>
<td>Encodes a specified string</td>
</tr>
<tr>
<td>MapPath()</td>
<td>Maps a relative or virtual path to the corresponding physical directory on a server</td>
</tr>
<tr>
<td>Transfer()</td>
<td>Sends the current ASP.NET application collection variables and objects to another ASP.NET application</td>
</tr>
</tbody>
</table>

---
field from StartPage.aspx. The form **action** attribute opens a SignGuestBook.aspx document that contains ASP.NET code that saves the form values in the guest book.

```html
<form method="post" action="SignGuestBook.aspx">
<p>
First Name: <input type="text" name="firstName" size="10" value="<%= Request.Form('first') %>">
Last Name: <input type="text" name="lastName" />
E-mail: <input type="text" name="email"/>
<input type="submit" value="I'm Finished" /></p>
</form>
```

4. Create another form at the end of the document body that opens a document named ShowGuestBook.aspx. The ShowGuestBook.aspx document will contain ASP.NET code that displays a list of the people who have signed the guest book.

```html
<form method="post" action="ShowGuestBook.aspx">
<p><input type="submit" value="See Who's Been Here" /></p>
</form>
```

5. Save **HomePage.aspx** and close it in your text editor.

Next, you will create the SignGuestBook.aspx document.

To create the SignGuestBook.aspx document:

1. Create a new document in your text editor.
2. Type the processing directive, as follows:
   ```html
   < %@ language="JScript" %>
   ```
3. Now type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “Guest Book” as the content of the `<title>` element.
4. Add the following code render block to the end of the document head. The first statement locks the application. Then an if statement with a logical not (!) operator checks if a **guestCounter** variable has been created in the **Application** object. The if statement creates the variable if it does not exist. The value of the **guestCounter** variable is then assigned to the **curGuestNum** variable, incremented by one, and then reassigned to the **guestCounter** variable. Another variable, **curGuestInfo**, is created to hold the first name, last name, and e-mail address of the guest signing the book. The statement `Application.Contents("guest" + curGuestNum) = curGuestInfo;` creates a new property in the **Application** object **Contents** collection to hold the current guest’s information. The last statement in the server section unlocks the application.
<% Application.Lock();
if (!Application.Contents("guestCounter"))
    Application.Contents("guestCounter") = 0;
var curGuestNum = Application.Contents("guestCounter");
curGuestNum = ++curGuestNum;
Application.Contents("guestCounter") = curGuestNum;
var curGuestInfo = Request.Form("firstName") + " "
    + Request.Form("lastName")
    + ", " + Request.Form("email");
Application.Contents("guest" + curGuestNum) = curGuestInfo;
Application.UnLock();%>

5. Add the following elements and form to the document body, which displays a customized message to the user, using the firstName property of the Request object Form collection. The form contains a single button that executes the history.back() method, which redisplays HomePage.aspx.

<p><strong>Thank you <%= Request.Form("firstName") %>! Your name and e-mail address have been added to our guest book.</strong></p>
<form action=""
<p><input type="button" value="Return to WebAdventure's Home Page" onclick="history.back();" /></p>
</form>

6. Save the document as SignGuestBook.aspx in your Chapter folder for Chapter 11 and close it in your text editor.

Finally, you will create the ShowGuestBook.aspx document.

To create the ShowGuestBook.aspx document:

1. Create a new document in your text editor.
2. Type the processing directive, as follows:
   
   < %@ language="JScript" %>

3. Now type the <!DOCTYPE> declaration, <html> element, header information, and the <body> element. Use the strict DTD and “Guest Book” as the content of the <title> element.
4. Add the following elements and text to the document body:

```html
<h2>This is the Guest List</h2>
<p><strong>Name, E-Mail</strong></p>
```

5. Add the following code render block to the end of the document body, which locks and unlocks the application, and uses a `for` loop to return the contents of each guest variable in the `Application` object `Contents` collection:

```html
<% Application.Lock();
var numGuests = Application.Contents("guestCounter");
for (var count = 1; count <= numGuests; ++count) {
    Response.Write(Application.Contents("guest" + count) + "<br />");
}
Application.UnLock();
%
```

6. Finally, add the following form to the end of the document body. The form contains a single button that executes the `history.back()` method to return to `HomePage.aspx`.

```html
<form action="">
<p><input type="button" value="Return to WebAdventure’s Home Page" onclick="history.back();" /></p>
</form>
```

7. Save the document as `ShowGuestBook.aspx` in your Chapter folder for Chapter 11 and close it in your text editor.

8. Open `StartPage.aspx` in your Web browser. Enter your name in the First Name text box and click the `Continue` button to open `HomePage.aspx`. Fill in your last name and e-mail address, and then click the `I’m Finished` button to sign the guest book. Figure 11-10 shows `SignGuestBook.aspx` in Internet Explorer after the user has entered a name and e-mail address.
9. Click the **Return to WebAdventure’s Home Page** button to return to HomePage.aspx, and then click the **See Who’s Been Here** button to view a list of people who have signed the guest book. Figure 11-11 displays ShowGuestBook.aspx after several guests have signed the book.

![Figure 11-11](ShowGuestBook.aspx)

10. Close your Web browser and text editor.
CHAPTER SUMMARY

- A system consisting of a client and a server is known as a two-tier system.
- A three-tier, or multitier, client/server system consists of three distinct pieces: the client tier, the processing tier, and the data storage tier.
- An important question in the design of any client/server system is deciding how much processing to place on the client.
- A Web application is a program that executes on a server but that clients access through a Web page loaded in a browser.
- ASP.NET is a server-side scripting language created by Microsoft.
- The term compiling refers to the processing and assembly of programming code into an executable format.
- The term ASP.NET application refers to a collection of related ASP.NET documents that exist in the same root directory.
- You define ASP.NET global variables and functions within code declaration blocks, which you create with `<script>` elements, the same as client-side JavaScript.
- A delimiter is a character or a sequence of characters used to mark the beginning and end of a code segment.
- A code render block executes ASP.NET code within a Web page document.
- An ASP.NET processing directive provides a Web server with information on how to process the scripts in an ASP.NET document.
- The output directive sends the result of an expression to a user’s Web browser (the client).
- Collections are data structures similar to arrays that store variables in ASP.NET core objects.
- The Request object represents the current URL request from a client.
- The Response object sends output and information back to the client.
- The Session object temporarily stores specific client information and makes that information available to all the pages in an ASP.NET application.
- An Application object is used for storing global application information that can be shared by all clients accessing the application.
- The Server object provides an ASP.NET application with access to properties and methods that are accessible by all applications on the server.
**Review Questions**

1. A system consisting of a client and a server is known as a _____________.
   a. mainframe topology
   b. double-system architecture
   c. two-tier system
   d. wide area network

2. What is usually the primary role of a client?
   a. locating records that match a request
   b. heavy processing, such as calculations
   c. data storage
   d. the presentation of an interface to the user

3. Which of the following functions does the processing tier **not** handle in a three-tier client/server system?
   a. processing and calculations
   b. reading and writing of information to the data storage tier
   c. the return of any information to the client tier
   d. data storage

4. Which function can a client safely handle?
   a. data validation
   b. data storage
   c. intensive processing
   d. heavy calculations

5. You add ASP.NET code to a document using which of the following elements? (Choose all that apply.)
   a. `<asp>...</asp>`
   b. `<%...%>`
   c. `<script>...</script>`
   d. `<aspserver>...</aspserver>`

6. Which is the correct syntax for creating a processing directive that sets the default scripting language to JScript?
   a. `<% lang="JScript" %>`
   b. `<%@ default="JScript" %>`
   c. `<% language="JScript" %>`
   d. `<%@ language="JScript" %>`
7. Which is the correct syntax for including the ASP.NET `Write()` method inside an element?
   a. `<% Write("Hello World"); %>`
   b. `<%= Write("Hello World"); %>`
   c. `<% Response.Write("Hello World"); %>`
   d. `<%= Response.Write("Hello World"); %>`

8. Which collection is found in both the `Application` and `Session` objects?
   a. Contents
   b. Preferences
   c. Cookies
   d. Properties

9. You can include function declarations inside code render blocks. True or false?

10. Which object contains the `Form` collection?
    a. Request
    b. Response
    c. Session
    d. Application

11. Which of the following objects has the shortest lifespan?
    a. Request
    b. Response
    c. Session
    d. Application

12. How do you refer to a form field named “password,” using the `Request` object?
    a. Request.password
    b. Request.Form.password
    c. Request.Form("password")
    d. Request.Contents("password")

13. Which method is used for changing the lifespan of a `Session` object?
    a. the `timeout()` method
    b. the `Timeout` property
    c. the `lifespan` property
    d. the `Duration()` method
14. In which object would you store an incremented invoice number that must be accessible to all users who access the ASP.NET application?
   a. Request
   b. Session
   c. Application
   d. Server

15. Which method prevents other clients from accessing properties of the Application object or Server object?
   a. lock()
   b. Lock()
   c. frozen()
   d. Preserve()

---

**HANDS-ON PROJECTS**

**Project 11-1**

In this project, you will create an ASP.NET program that uses the `Response.Write()` statement to print “Print from an ASP.NET Program” to the screen.

1. Create a new document in your text editor.
2. Type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “Simple Output” as the content of the `<title>` element.
3. Add the following processing directives and code render block to the document body:
   ```html
   < %@ language="JScript" %>
   <% Response.Write("<p>Printed from an ASP program.</p>"); %>
   ```
4. Save the document as `SimpleOutput.aspx` in a folder named `SimpleOutput` in your Projects folder for Chapter 11 and then open it from your ASP.NET server. You should see the text written to the screen.
5. Close your Web browser window.
**Project 11-2**

In this project, you will create an ASP.NET program that uses the `Response.Write()` statement to print the names of five technology companies.

1. Create a new document in your text editor.
2. Type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “Technology Companies” as the content of the `<title>` element.
3. Add the following processing directives and code render block to the document body:

   `<%@ Language="JScript" %>
   <% 
   Response.Write("<p>Microsoft<br />
   Cisco<br />
   Lucent<br />
   Sun Microsystems<br />
   Oracle</p>"select>
   %>

4. Save the document as `TechCompanies.aspx` in a folder named `TechCompanies` in your Projects folder for Chapter 11 and then open it from your ASP.NET server. You should see the five lines written to the screen.
5. Close your Web browser window.

**Project 11-3**

In this project, you will create an ASP.NET program that uses the `Response.Write()` statement to print six lines; each line is formatted using one of the six heading-level elements.

1. Create a new document in your text editor.
2. Type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “Heading Elements” as the content of the `<title>` element.
3. Add the following processing directives and code render block to the document body:

   `<%@ language="JScript" %>
   <% 
   Response.Write("<h1>Heading 1</h1>"select>
   Response.Write("<h2>Heading 2</h2>"select>
   Response.Write("<h3>Heading 3</h3>"select>
   Response.Write("<h4>Heading 4</h4>"select>
   Response.Write("<h5>Heading 5</h5>"select>
   Response.Write("<h6>Heading 6</h6>"select>
   %>

4. Save the document as `TechCompanies.aspx` in a folder named `TechCompanies` in your Projects folder for Chapter 11 and then open it from your ASP.NET server. You should see the six lines written to the screen.
5. Close your Web browser window.
4. Save the document as **Headings.aspx** in a folder named **Headings** in your Projects folder for Chapter 11 and then open it from your ASP.NET server. You should see the five lines written to the screen.

5. Close your Web browser window.

### Project 11-4

In this project, you will create an ASP.NET program that uses the `Response.Write()` statement to print lines that use various text-formatting elements.

1. Create a new document in your text editor.

2. Type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “Text-formatting Elements” as the content of the `<title>` element.

3. Add the following processing directives and code render block to the document body:

   ```
   <%@ language="JScript" %>
   <%
   Response.Write("<p><b>Bold text</b><br/>");
   Response.Write("<i>Italicized text</i><br/>");
   Response.Write("<big>Big text</big><br/>");
   Response.Write("<small>Small text</small><br/>");
   Response.Write("<sub>subscripted</sub>&nbsp;text<br/>");
   Response.Write("<tt>Teletype text</tt><br/>");
   Response.Write("<sup>superscripted</sup>&nbsp;text</p>";)
   %>
   ```

4. Save the document as **TextFormats.aspx** in a folder named **TextFormats** in your Projects folder for Chapter 11 and then open it from your ASP.NET server. You should see the five lines written to the screen.

5. Close your Web browser window.

### Project 11-5

In this project, you will create an ASP.NET script that assigns the value of a global variable as the value of an `<a>` element’s `href` attribute.

1. Create a new document in your text editor.

2. Type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “Mail Variable” as the content of the `<title>` element.
3. Add the following processing directives and code declaration block to the document head. Replace the value assigned to the myEmail variable with your personal e-mail address.

```html
<script language="JScript" runat="server">
    var myEmail = "your e-mail address";
</script>
```

4. Add the following elements and text to the document body:

```html
<h2>ASP.NET Variable</h2>
<p><a>Click here to send an e-mail</a>.</p>
```

5. Modify the `<a>` element so it references the myEmail variable. After ASP.NET returns the document to the user, the user should be able to click the link and send you a message.

6. Save the document as `MailVariable.aspx` in a folder named `MailVariable` in your Projects folder for Chapter 11 and then open it from your ASP.NET server and click the link. Depending on how your computer is set up, the software you use to send e-mail should open a new message addressed to you.

7. Close your Web browser window.

### Project 11-6

In this project, you will create an ASP.NET script that welcomes the user to the Web page and displays the current date and time.

1. Create a new document in your text editor.

2. Type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “Welcome” as the content of the `<title>` element.

3. Add the following processing directives and code render block to the document body:

```html
<%@ language="JScript" %>
<% 
    var dateObject = new Date();
    var month = dateObject.getMonth();
    var monthArray = new Array("January","February","March","April","May","June",
    "July","August","September","October","November","December");
    Response.Write("<h1>Welcome!</h1>");
    Response.Write("<p>The date is " + monthArray[month] + " "
    + dateObject.getDate() + "", " + dateObject.getFullYear()
    + "<br />"");
    Response.Write("<p>The time is " + dateObject.getHours()
```
Hands-on Projects

4. Save the document as Welcome.aspx in your Projects folder for Chapter 11. Open the Welcome.aspx document from your ASP.NET server. You should see the greeting along with the current date and time.

5. Close your Web browser window.

Project 11-7

In this project, you will create an ASP.NET program that returns a response to the user containing information that was submitted in a form.

1. Create a new document in your text editor.

2. Type the <!DOCTYPE> declaration, <html> element, header information, and the <body> element. Use the strict DTD and “Profession” as the content of the <title> element.

3. Add the following heading and form to the document body:

   <h2>Profession</h2>
   <form method="post" action="ProfessionScript.aspx">
   <p>First name: <input type="text" name="firstName" size="50" /></p>
   <p>Last name: <input type="text" name="lastName" size="50" /></p>
   <p>Profession: <input type="text" name="profession" size="50" /></p>
   <p><input type="submit" /> <input type="reset" /></p>
   </form>

4. Save the document as ProfessionForm.html in a folder named Profession in your Projects folder for Chapter 11.

5. Create another new document in your text editor.

6. Type the <!DOCTYPE> declaration, <html> element, header information, and the <body> element. Use the strict DTD and “Profession” as the content of the <title> element.

7. Add the following processing directives and code render block to the document body:

   < %@ language="JScript" %>
   <%
   Response.Write("<p>You submitted the following information:</p>");
   Response.Write("<p>First name:&nbsp;" + Request.Form("firstName") + "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&ndash;11
Response.Write("Last name: 
  + Request.Form("lastName") + "<br />");
Response.Write("Profession: 
  + Request.Form("profession") + "</p>");

8. Save the document as **ProfessionScript.aspx** in your Projects folder for Chapter 11 and then open the **ProfessionForm.html** document from your ASP.NET server. Enter some data in the form fields and then click the Submit button. You should see the values you entered returned by the ASP.NET script.


**Project 11-8**

In this project, you create a Web page with a form that lists several sports. Clicking the form’s Submit button returns a response from an ASP.NET script that determines where the selected sport is played.

1. Create a new document in your text editor.

2. Type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “Sports” as the content of the `<title>` element.

3. Add the following paragraph and form to the document body:

   ```html
   <h2>Sports</h2>
   <p>What is your favorite sport?</p>
   <form method="post" action="SportsScript.aspx">
   <p><input type="radio" name="sport" value="golf" checked="checked" /> Golf<br />
   <input type="radio" name="sport" value="tennis" /> Tennis<br />
   <input type="radio" name="sport" value="baseball" /> Baseball<br />
   <input type="radio" name="sport" value="basketball" /> Basketball</p>
   <p><input type="submit" /> <input type="reset" /></p>
   </form>
   ```

4. Save the document as **SportsForm.html** in your Projects folder for Chapter 11.

5. Create another new document in your text editor.

6. Type the `<!DOCTYPE>` declaration, `<html>` element, header information, and the `<body>` element. Use the strict DTD and “Sports” as the content of the `<title>` element.
7. Add the following processing directives and code render block to the document body:

```html
<%@ Language="JScript" %>
<%
if (Request.Form("sport") == "golf")
Response.Write("<p>Golf is played on a golf course.</p>"));
else if (Request.Form("sport") == "tennis")
Response.Write("<p>Tennis is played on a tennis court.</p>"));
else if (Request.Form("sport") == "baseball")
Response.Write("<p>Baseball is played on a baseball diamond.</p>"));
else if (Request.Form("sport") == "basketball")
Response.Write("<p>Basketball is played on a basketball court.</p>"));
%>
```

8. Save the document as `SportsScript.aspx` in your Projects folder for Chapter 11 and then open the `SportsForm.html` document from your ASP.NET server and select a sport. Click the `Submit` button. You should receive a response that says where the sport is played.


### CASE PROJECTS

For the following projects, save the documents you create in your Cases folder for Chapter 11.

#### Case Project 11-1

Create a generic ASP.NET redirection page that you can use to send clients to pages other than the one they requested. You can add this page to your library of JavaScript tools. Use the ASP.NET `Response` object `redirect()` method. Save the document as `Redirection.aspx`.

#### Case Project 11-2

Create an ASP.NET program that allows users to play a guessing game. Use the JavaScript `Math` object to create a random number between 0 and 100. Assign the number to a property of the `Session` object. Include a form with a text field where the user can enter a number, and add a button named `Guess`. When the user enters a number and clicks the `Guess` button, another page should open and score the test. The scoring page should display “You guessed to high!” if the number is too high and “You guessed too low!” if the number is too low. Record the number of tries in the `Session` object. Save the main document as `GuessingGame.aspx` and the scoring page as `CheckNumber.aspx`. 
Case Project 11-3

Create an ASP.NET program for tracking, documenting, and managing the process of interviewing candidates for professional positions. Name the first page in the program Interview.html, and include a form with fields for the interviewer’s name, position, and the date of the interview. Also include fields such as candidate’s name, communication abilities, professional appearance, computer skills, business knowledge, and the interviewer’s comments. Clicking the Submit button should submit the form to a document named SubmitApp.aspx, which saves the data in the Application object. The Interview.html document should also include a form with a single Submit button that opens a document named Candidates.aspx. The Candidates.aspx should display a list containing the candidate’s name, along with an identification number for each candidate. The Candidates.aspx document should also contain a form that opens a document named CandidateInfo.aspx, which allows interviewers to display each candidate’s interview information.

Case Project 11-4

Create an ASP.NET program for recording company invoices. Name the first page in the program Invoice.html. The Invoice.html document should contain two buttons, New Invoice and Review Invoices. The New Invoice button should open a document with a form that contain fields you would find on an invoice, including customer name, shipping address, item descriptions, quantities, and prices. Clicking the form’s Submit button should submit the form to a document named RecordInvoice.aspx, which saves the data in the Application object. The Review Invoices button on the main page should open a document that displays the list of invoices by customer name, along with the invoice number. The ReviewInvoices.aspx document should also contain a button that opens a document named InvoiceInfo.aspx, which displays the information entered into each invoice.