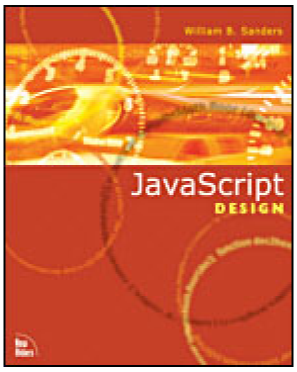
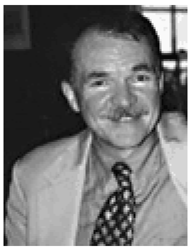
JavaScript Design

William B. Sanders
Publisher: New Riders Publishing
First Edition December 20, 2001
ISBN: 0-7357-1167-4, 600 pages



JavaScript Design

About the Author



Dr. William B. Sanders is a professor in the Interactive Information Technology program at the University of Hartford. The program is designed to develop students who will work in collaborative environments using the Internet and the World Wide Web and develop digital communicative technologies. Bill has written more than 35 computer-related books, with the goal of translating technology to a wide interest base. To mangle a phrase from Will Rogers, he never met a computer or computer language that he didn't like.

Like the revolution spawned by personal computers, the Internet and the World Wide Web have spawned another. The new languages and applications required to master and effectively use Internet technologies have been a focal interest of Bill's since the web's inception. He has been focused on languages such as JavaScript, PHP, ASP, XML, ActionScript, MySQL, and a host of other web-based programs and applications. However, instead of looking at the new technologies solely as a cool way to make things happen on the web, Bill has been involved with different aspects of e-business and e-commerce, bridging the digital divide in communities and generally looking at ways in which the Internet and the web serve as a lively linkage between people and their aspirations.

As a source of information and understanding, the web is unparalleled, but it is also an arena to explore new art forms and ways of human expression. Bill has sought out design concepts from Edward Tufte's work on information, Hillman Curtis's work on motion design, and David Siegel's work on third-generation web sites. For Bill, each new development in creativity, technology, and communication is an opportunity to see the world in a new light and expand horizons.

His hobbies include travel, Greater Swiss Mountain Dogs, and life with his wife, Delia.

About the Technical Reviewers

These reviewers contributed their considerable hands-on expertise to the entire development process for *JavaScript Design*. As the book was being written, these dedicated professionals reviewed all the material for technical content, organization, and flow. Their feedback was critical to ensuring that *JavaScript Design* fits our readers' need for the highest-quality technical information.

Josh Kneedler resides in Portland, Oregon. He is a founding partner of the visual media studio Dreaming America (http://dreamingamerica.com). With the support of Dreaming America, Josh has also started an online magazine called *Rangermag* (http://rangermag.com). Over the years since 1997, Josh has acquired a strong sense of both functionality and design. He can be reached at josh@dreamingamerica.com.

Joel Lee and **Bryan Ginz** are technical editors for JTL Networks, Inc. (JTLNET). Based in Columbus, Ohio, JTLNET provides a variety of information technology and consulting services for businesses, including managed web hosting, custom programming and design, remote administration, remote data storage, and network design. For more information on JTLNET, visit www.jtlnet.com.

Acknowledgments

This book began back in 1996 using JavaScript 1.1 and later JavaScript 1.2, when it became available. A group of us at the University of Hartford got together once a week to create problems to be solved with JavaScript and, much to my surprise and delight, many of the tools, applications, and utilities that we developed then, we still use today. Of those involved, the brightest of this group was and remains to be David Kelly. Dave developed a Quiz-Maker in JavaScript that still makes great online quizzes. He also seemed to be about five jumps ahead of the rest of us and was a great help to us all. Laura Spitz, a designer *extraordinaire,* still does extraordinary designs and uses JavaScript regularly in her work. She introduced me to BBEdit and HomeSite, which have yet to replace NotePad and SimpleText on my PC and Mac, respectively. Finally, Morris Hicks, who recently took over the Assistant Director of the Office of Information Technology position at Boise State University, was a regular with our group and provided a knowledgeable presence at our meetings.

Also at the University of Hartford, I'd like to thank the faculty and students in the Interactive Information Technology program. The students are an always creative lot who challenge the faculty to come up with better courses, answers, and scripts. Likewise, the IIT faculty, including John Gray, Jerry Katrichis, and David Demers, are a good group to kick ideas around with. Also, Steve Misovich and Lou Boudreau of the University of Hartford Hillyer College psychology faculty were a true inspiration as we developed a virtual psychology lab for the web. Everything from a rat maze to a timed reaction experiment were accomplished using JavaScript as part of a grant application to the National Science Foundation. During this project, I learned that there is very little that cannot be accomplished with JavaScript once a project goal has been set.

Next, I'd like to thank the people at New Riders. Thanks to Stephanie Wall, who worked with me to develop an outline for the book that would focus on designers' needs in working with and understanding JavaScript. Also, I would like to thank John Rahm for helping to develop everything just right and Jake McFarland for coordinating the details of the finishing touches. Thanks also to the copy and technical editors—Krista Hansing, Josh Kneedler, Joel Lee, and Bryan Ginz— for locating the glitches and setting them straight.

Finally, I'd like to thank my wife, Delia. As I was starting this book, she was beginning her doctoral work at Smith College, and so both of us were in our studies, thinking and writing together. Like everything else we do together, it brought us closer.

Tell Us What You Think

As the reader of this book, you are the most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As the Associate Publisher for New Riders Publishing, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax:	317-581-4663
Email:	stephanie.wall@newriders.com
Mail:	Stephanie Wall Associate Publisher New Riders Publishing 201 West 103rd Street Indianapolis, IN 46290 USA

Part I: Basic JavaScript

Chapter 1. Jump-Starting JavaScript

CONTENTS>>

Getting Started With JavaScript is like beginning any other scripting, or programming, language. To learn it, you have to use it. JavaScript is the "engine" that makes things move on a page; by working with dynamic design elements, the more you see what can be done with JavaScript and the more incentive there is to learn to use it. JavaScript allows designers to release those aspects of design creativity that cannot be expressed in static HTML.

You need not look very far to find a use for JavaScript, and so opportunities abound for learning the language—rollovers, moving text, prompt windows, and alert boxes are just a few of the actions powered by JavaScript. JavaScript is a *ranged* language. It ranges from extremely simple built-in functions and statements that can make your page jump to fairly sophisticated coding structures. By beginning with the simple, you can ease your way to its more complex and powerful structures as dictated by design needs. It doesn't require a compiler or a degree in computer science to learn. It lives right among the HTML tags, and most JavaScript programs are relatively small, so you're not spending all your life writing hundreds of lines of code.

Throughout this book, you will see explanations accompanied by examples and applications. However, the applications are really only extensions of some feature or concept in JavaScript. My goal with this book is not merely to enable you to cut and paste snippets of code, but rather to understand JavaScript in a way that you can apply to your own projects. When you master JavaScript, you will be able to imagine a project or design in your mind's eye and then create the JavaScript necessary to make your imagined scene a reality on a web page.

JavaScript captures user events that cause actions to happen on a web page. As a designer who has mastered JavaScript, you will be able to invent new ways that the user interacts with an interactive project.

JavaScript Lives in a Web Page

All the code that you write for JavaScript goes into an HTML page. If you don't know HTML yet, you should run out and get a good book on HTML. Lynda and William Weinman's *Creative HTML Design.2* (New Riders, 2001) is a good choice for designers and developers. However, assuming that you are familiar with HTML, you should be familiar with the whole concept of a tag language. HTML stands for Hypertext Markup Language. As a markup language, HTML essentially *describes* a web page as a static entity. A far more challenging endeavor is to program a web page that is dynamic, engaging, and intriguing. That's where JavaScript comes into play.

The most dynamic elements in HTML, beside the link, are event-related attributes. For example, `onClick` is one of the event-related attributes of HTML. The HTML attribute launches a script when the user clicks on a portion of the page sensitive to a mouse-click action set up by the HTML. However, because HTML itself has no dynamic components, it relies on scripts written in JavaScript. An event-related attribute in HTML is like having a starter on a car with no engine—JavaScript is the engine.

When you have finished an HTML page using solely HTML, the page sits on the screen until you click a link that connects to a separate HTML page, which makes the current page go away.

With JavaScript, you can create pages that make something happen on the page when the person viewing the page takes an action that fires a JavaScript. For example, you might have seen pages that have buttons that change shape or color when the mouse passes over them. That change can be made with a script written in JavaScript and fired by an event-related attribute in HTML: `onMouseOver`. You are also working on a page that doesn't necessarily have to make server requests. All the interaction is taking place without having to download anything. Depending on the application, this can set the groundwork for instantaneous responsive experiences.

Putting JavaScript into Your HTML Pages

This section contains several scripts written in JavaScript that illustrate some of the things that you can do using JavaScript that cannot be done with HTML alone. All these scripts are simple, but at this point in the book, they are not explained beyond the most general terms. They serve to illustrate where to place JavaScript code and the difference between immediate and deferred working of the script. An immediate script executes as soon as it loads, and a deferred script waits until the user does something to make the script launch.

Most JavaScript is written in a tag container named script. The generic format looks like the following:

```
<script language="JavaScript">
script goes here
</script>
```

As you will see in the examples throughout this book, the script container is required for *most* JavaScript, even though a few cases exist in which JavaScript can be applied on the fly. However, you should expect to see the opening and closing `<script>` tags where you see JavaScript.

CAUTION

Unlike some tags that do not require an ending or closing tag, the `<script>` *absolutely requires a* `</script>` *tag. In debugging your script, the first thing to check is to make sure that you put in both tags.*

For example, the following is a simple, minimal script of what an HTML page needs for JavaScript to work:

```
<html>
<body>
<script language="JavaScript">
document.write("Hello designers and developers.");
</script>
</body>
</html>
```

The JavaScript container has a single line of JavaScript to be executed as soon as the parser passes over it. The parser is the interpreter that reads the code one line at a time, beginning with the top line. Usually, you will find JavaScript code in the head section of an HTML page. (The area in the `<head>…</head>` container is the head.) All the code in the head section of an HTML page is loaded first; if code is placed in the head, you do not have to worry about the code being only partially loaded when the viewer is ready to fire the JavaScript functions.

What You Can Do with JavaScript That You Can't Do with HTML

The most important feature that JavaScript can add to a web site design is the capability to introduce *dynamic interactivity* into pages. The concept of dynamic interactivity implies change in response to an action. For example, a design might be one that seeks to engage the viewer in the page. By having page elements that respond to the viewer's action, the designer can have a page that interacts with the viewer rather than just sitting there for the viewer to look at. In one respect, HTML sites are interactively dynamic because the viewer navigates to different places in the site depending on where she clicks on the page. However, the pages themselves are fairly static. With dynamic interactivity on a page, features of the page change as the user moves, clicks, or drags the mouse over the page.

Alerting the Viewer

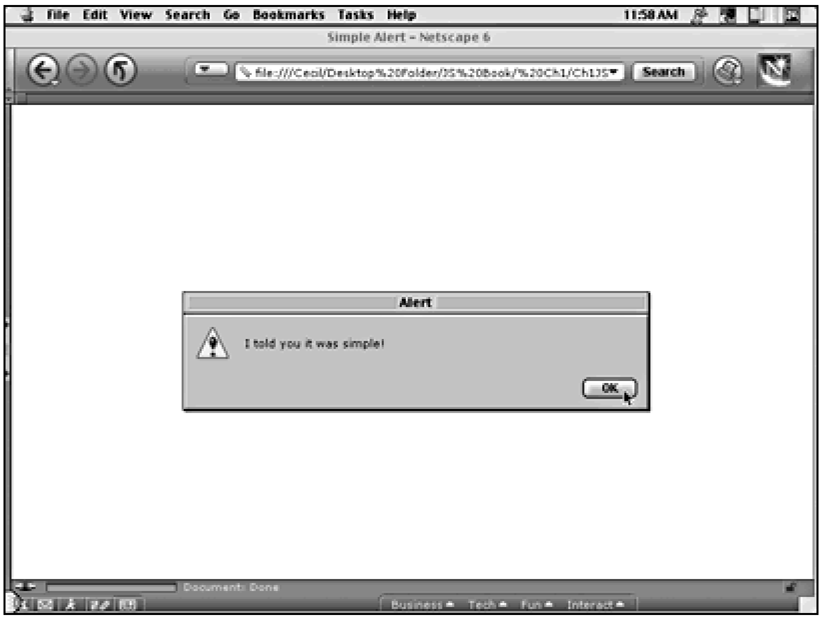
A useful built-in function in JavaScript is the `alert( )` function. This function sends a message to the page. The contents of the message can vary depending on what the user does, or the messages can be static. When the alert box appears with the message, the user clicks to close it. The following example is a very simple one that illustrates how to use a function in a script. Throughout the book, examples using the `alert( )` function will help you understand JavaScript, and I think you will find it a good learning tool. Of course, you can use it in more sophisticated ways than what is presented here. When you start writing more complex functions, you will find the `alert( )` function valuable in terms of pinpointing problem areas in the function.

Later in the book, you will learn a lot more about functions. However, for now, it is enough to know that JavaScript contains built-in functions that execute a set of instructions to do things like put messages on the screen, prompt users to enter information, and write text to the screen. You may also write your own functions consisting of a set of statements and other functions. Typically, a function's actions are deferred until the user does something, such as click a button, to launch them. However, as you will see in the following script, a function can be fired in the immediate mode.

```
<html>
<head>
<title>Simple Alert </title>
</head>
<body>
<script language="JavaScript">
alert("I told you it was simple!");
</script>
</body>
</html>
```

The only dynamic interactivity that the user engages is clicking the OK button to remove the message from the screen. However, later in the book, you will see some far more dynamic uses of this handy little function. Figure 1.1 shows what you should see when your page comes up.

Figure 1.1. JavaScript adds interactivity to web pages.


File Edit View Search Go Bookmarks Tasks Help
11:58 AM
Simple Alert - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/%20Ch1/Ch1JS
Search
Alert
OK
Document: Done
Business Tech Fun Interact

I told you it was simple!

Prompting a Reaction

The second example of dynamic interactivity on a page using JavaScript can be seen using the `prompt( )` function. This built-in function can take two arguments. An *argument* is some value in the function that you can change to different values, whether it is entered in the code itself or whether the user puts it in himself. Thus, a single function can have several different values throughout a single script, and you can use it with several different designs. All you need to change are the arguments in the function; you don't have to rewrite the function even though the design is different. A Cascading Style Sheet (CSS) is added to provide an integrated color system to the page. Chapter 12, "Dynamic HTML," goes into detail about using CSS, but I believe that you should start using CSS in all HTML designs. CSS soon will be replacing tags such as `<font>` for specifying colors in text, and CSS helps designers better set up color schemes, font sizes and weights, and overall design of a page. Also, you can see how CSS can be integrated into a JavaScript program.

prompt.html

```
<html>
```

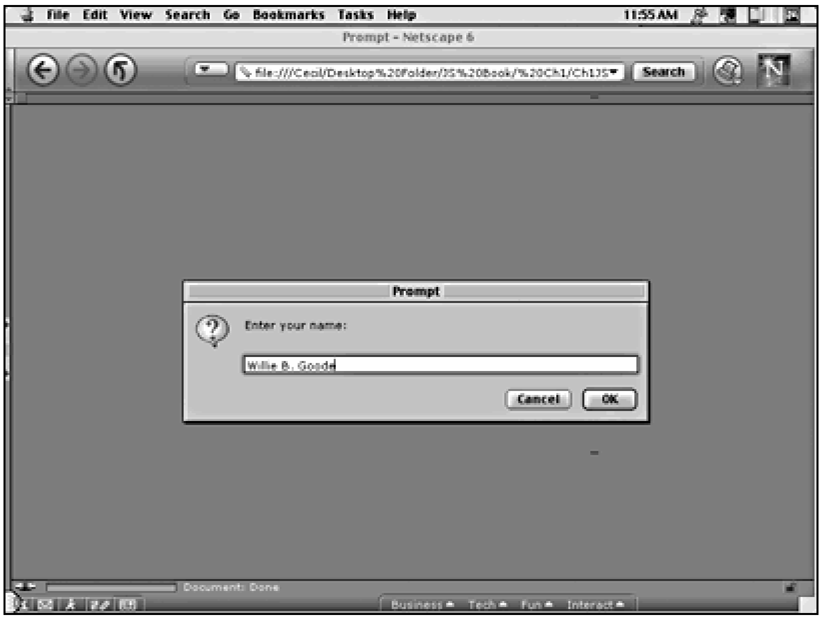
```
<head>
<title>Prompt</title>
<style type="text/css" >
.prompt {
font-family: verdana;
font-weight:bold;
color: #8b6952;
background-color:#d3c75e;
font-size:18pt
}
body {
background-color: #db4347
}
</style>
</title>
<body>
<center><p><p>
<div class=prompt>
<script language="JavaScript">
var yourname
yourname=prompt("Enter your name:","Name");
document.write("Welcome, " + yourname);
</script>
</div>
</body>
</html>
```

CAUTION

If you use a version other than Version 6 of Netscape Navigator, you will see a different background when using Netscape Navigator and Internet Explorer. Prior to Version 6, the browsers interpreted CSS differently, but current browsers adhere to the standards set by the World Wide Web Consortium (W3C).

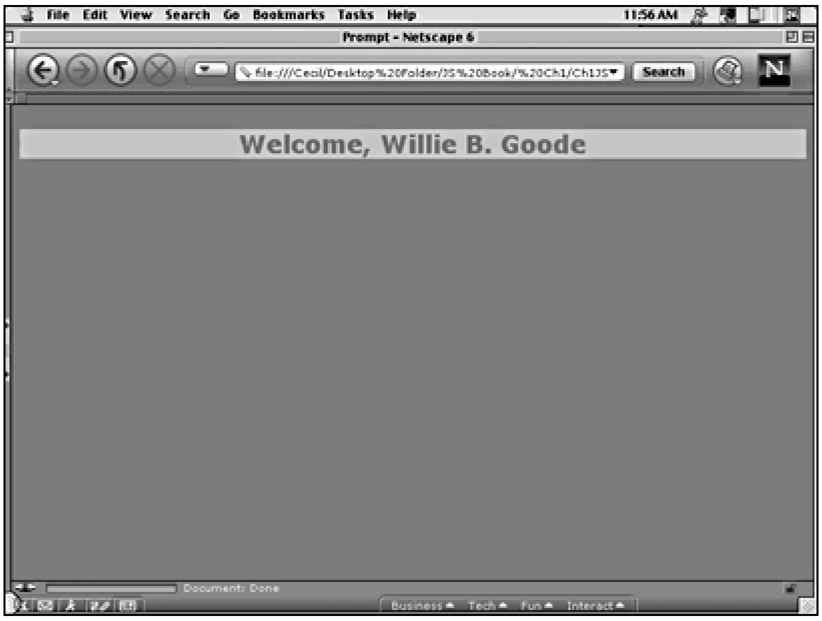
The two arguments used in the `prompt( )` function are the prompt message that you select (Enter Your Name) and the optional placeholder that appears in the prompt window (Name). When the user opens the page, the script automatically runs. The script is right in the middle of the HTML and is executed as soon as the parser reads and interprets the code. Figure 1.2 shows what the viewer sees when the page opens.

Figure 1.2. The viewer is prompted to enter a name.



After the viewer has entered information, he sees his name on the web page, as shown in Figure 1.3.

Figure 1.3. The information entered by the user now appears on the web page.



Changing Background Colors

The capability to change background colors on the fly allows the designer to get the viewer’s attention. First, whenever a background color changes, the viewer is alerted to *something* different going on, and this change can be used to produce a mood or context that the designer wants to impart. Second, creative designers have used changes in background color to reveal hidden text. White text on a white background is invisible, but as soon as the background turns to black, the white text is revealed and black text is hidden.

This next example shows one of the few places where JavaScript is not required to have a container to first define JavaScript and an example of JavaScript running in the deferred mode. The buttons in the form serve as firing mechanisms for the JavaScript code. No JavaScript is executed until a button is pressed. Unlike the previous two examples, the JavaScript in this script is *deferred.*

This line:

```
<input type="button" value="#d0d0a9"
onClick="document.bgColor='d0d0a9'">
```

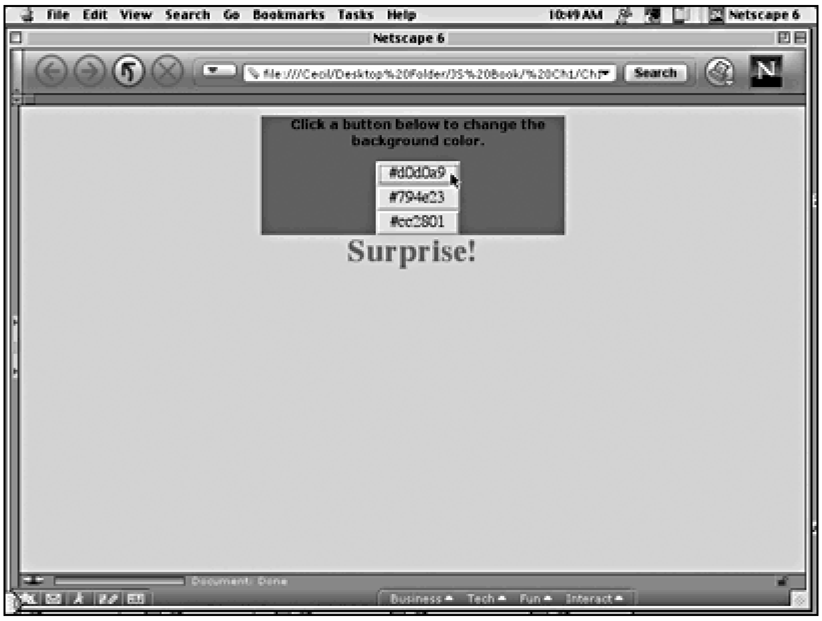
contains the firing button (`"button"`), the event handler attribute (`onClick`), and the built-in JavaScript (`document.bgColor=`). The CSS code takes up all the room in the head of the page; however, it is important because it not only defines the body text, but also the masked message that appears when the background color changes to reveal text on the page.

colorChange.html

```
<html>
<head>
<style type="text/css">
.bText {
font-family: verdana;
font-weight:bold;
font-size:10pt
}
.surprise {
font-weight:bold;
color: #cc2801;
font-size:24pt
}
</style>
</head>
<body bgcolor="#cc2801">
<center>
<table border=0 cellpadding=5 cols=1 width="40%" height="40%" >
      <tr>
           <td align=center valign=center bgcolor="#3c6816">
           <p class=bText>Click a button below to change the
background color.</p>
           <form>
<input type="button" value="#d0d0a9"
onClick="document.bgColor='d0d0a9'">
<br>
<input type="button" value="#794e23"
onClick="document.bgColor='#794e23'">
<br>
<input type="button" value="#cc2801"
onClick="document.bgColor='#cc2801'">
           </td>
      </tr>
</table>
<div class=surprise>Surprise!</div>
</center>
</body>
</html>
```

The preceding listing might seem to be a lot of code for changing background colors, but most of the code relates to the CSS—only three lines use the JavaScript. With all the changes that take place on the page, it appears as though several different pages are rapidly sequencing through the browser. However, it's just a single page with a little JavaScript. Figure 1.4 shows what the viewer will see when she changes background colors from the original.

Figure 1.4. By changing background colors, masked messages appear.



A Function to Convert Decimal to Hexadecimal

The heart and soul of JavaScript belongs in functions. In Chapter 6, "Building and Calling Functions," functions are discussed in detail. As noted previously, functions are self-contained clusters of JavaScript. The functions can then be used throughout the page in which they are defined and can be launched using event-related attributes. Furthermore, keep in mind that functions are either built-in, like the `alert( )` and `prompt( )` functions, or *user-defined.* A user-defined function is one that the programmer writes and uses in her page. However, the built-in and userdefined functions work and are launched in the same way.

This next script uses a function that converts three sets of decimal numbers into a hexadecimal number. While relatively complex for the first chapter, the purpose of the script is to show you something very practical that you can do with JavaScript. To get an exact match between colors that you develop in graphic programs such as Adobe Photoshop and your web page, you need to change the decimal RGB (red, green, blue) colors into a hexadecimal value. For example, a rich red has the following values:

R=169 G=23 B=35

To use those numbers, they have to be translated into a six-character hexadecimal value that you can see in the following CSS definition:

```
Body { background-color: a91723;}
```

You can find lots of decimal-to-hexadecimal converters on the web, but because you want to convert colors, you need the conversion in groups of three for the red, green, and blue values. Taking the three decimal values, your converter should convert the colors into a single six-character hexadecimal value. The following program does just that.

NOTE

If you're new to programming, do not be intimidated by the code in this next script. In the following several chapters, you will be introduced gradually to the kind of code and structures that allow you to build more sophisticated JavaScript programs. This one is simply an example of what you can do with JavaScript and one that you can use to convert decimal values into hexadecimal ones for use in your pages.

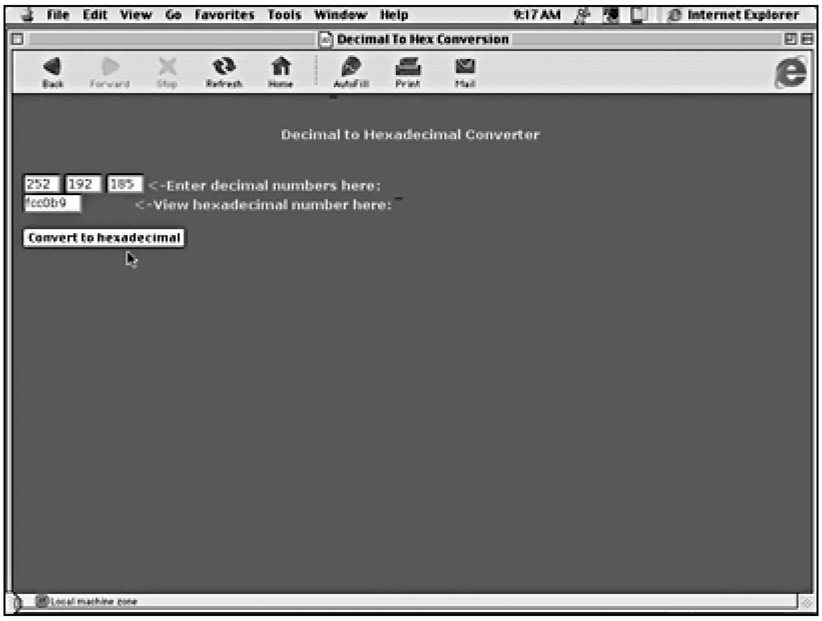
decimal2hex.html

```
<html>
<head>
<style>
body {
font-family: verdana;
background-color: a91723;
color: fcc0b9;
font-weight:bold;
font-size:10pt
}
</style>
<title>Decimal To Hex Conversion</title>
<script language="JavaScript">
function dec2hex( ) {
var accum=" ";
for (var i=0;i<3;i++) {
      var dec=document.convert.elements[i].value;
      var toNum=Math.floor(dec);
      var toHex=toNum.toString(16);
             if (toHex.length==1) {
             toHex= "0"+toHex;
             }
      accum += toHex;
      }
document.convert.hex.value=accum;
}
</script>
</head>
<body >
<center><br>
Decimal to Hexadecimal Converter
</center><br>
<form name="convert">
<input type="text" name="dec1" size=3>
<input type="text" name="dec2" size=3>
<input type="text" name="dec3" size=3>
<-Enter decimal numbers here:<br>
<input type="text" name="hex" size=6>
&nbsp; &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<-View hexadecimal number here:<p>
```

```
<input type="button" value="Convert to hexadecimal"
onClick="dec2hex( )">
</p>
</body>
</html>
```

The big line of `&nbsp` (nonbreaking spaces) is simply for adding spaces—a formatting chore. The script uses data entered by the web page viewer in the form windows. The function converts each decimal number using a loop, checking for single-digit results that require a leading zero (0), and then builds a string containing six characters that make up the hexadecimal values. Finally, the function sends the results to the viewer. Figure 1.5 shows you what the viewer sees when she enters values to be converted into a six-digit hexadecimal value.

Figure 1.5. JavaScript can create very practical and useful applications.


File Edit View Go Favorites Tools Window Help
9:17 AM
Internet Explorer
Decimal To Hex Conversion
Back
Forward
Stop
Refresh
Home
AutoFill
Print
Mail
Decimal to Hexadecimal Converter
252
192
185
<-Enter decimal numbers here:
#fcc0b9
<-View hexadecimal number here:
Convert to hexadecimal
Local machine zone

Now that you have seen some examples of what JavaScript can do in collaboration with HTML, the next sections provide some more information about how JavaScript works in your browser.

An Interpreted Language

Some languages are *interpreted,* and some are *compiled.* JavaScript is an interpreted language. Your browser, most likely Netscape Navigator or Internet Explorer, acts as a translator between JavaScript and the native language that your computer uses. The process involves your browser parsing (interpreting) the JavaScript code and then creating an equivalent machine language and having your computer execute the interpreted code. Compiled languages such as Java and C++ are already interpreted (compiled), so they go right into the computer, ready to talk with your computer.

The process is something like visiting France and communicating in French. If you don't know French, the easiest way to communicate is with an interpreter who speaks both French and English. It takes a little longer because everything that you say has to be translated into French. Alternatively, you could go to classes to learn to speak French before your visit to France. Learning French will take more time and effort, but when you go to France, you will be understood without an interpreter and communication will go much more quickly.

Interpreted and compiled languages in computers work somewhat along the same lines. Generally, interpreted languages are easier to learn and use, but they take more time to run in your computer. Compiled languages take more time to master, debug, and write, but they execute far more quickly. However, most JavaScript applications are fairly short. The interpreter doesn't have much to interpret—good day (*bonjour*)—and so the difference between a complied and interpreted language is often negligible. This is especially true today with the high speeds of modern computers.

A Tale of Two Interpreters

A bit more serious problem exists between the two main interpreters, Netscape Navigator and Internet Explorer. Each has a slightly different way of translating JavaScript. Fortunately, only a few elements are translated differently, but, unfortunately, the differences can create major problems. The European Computer Manufacturer's Association (ECMA) sets standards for JavaScript, and both Netscape and Microsoft *generally* adhere to these standards. However, for some reason, each has decided to have a slightly different way of interpreting JavaScript; when those differences are encountered, you need to know how to deal with them. In the meantime, be aware that JavaScript's interpretation of a few commands and statements has slightly different structures for the two competing interpreters.

NOTE

To make matters more interesting, both of the major browsers keep improving on their products. At this writing, Netscape Navigator (NN) is part of Version 6 of Netscape Communicator, and Internet Explorer (IE) is in Version 5.5. However, the numbers don't tell us much because NN skipped Version 5 altogether and went from Version 4.7 to Version 6. What is important is that the browsers are interpreters and that the interpreters determine what version of JavaScript each can read. Even though JavaScript 1.3 and 1.5 language elements are available, they're still in testing. Realistically, JavaScript's big developmental change came with JavaScript 1.2. While this book covers the new features added with JavaScript 1.3 and 1.5, most JavaScript in its newest configuration was present when JavaScript 1.2 appeared. The official revision version of JavaScript is ECMA-262, and JavaScript 1.2, 1.3, and 1.5 adhere to ECMA-262—with the exceptions that the browser manufacturers add. When JavaScript 2.0 is complete, you won't have to learn JavaScript all over again. The goal is to have backward compatibility with earlier versions of JavaScript. So, learning JavaScript, where most of the revisions were put into JavaScript 1.2 , is a safe starting place for now and for later revisions.

In the meantime, don't be overly concerned about all these different versions of JavaScript. Just be aware of them. If you use Version 4 and above on either of the major browsers, your JavaScript can be read just fine except where little differences of interpretation exist. You will be alerted to those places where the two major browsers part company and how to deal with the differences.

Generated JavaScript

Many designers have their first developmental encounter with JavaScript when they create web pages with tools such as Macromedia Dreamweaver, Adobe GoLive, or Microsoft FrontPage. Not only will these tools generate JavaScript for you, but they will do it for either or both of the major browsers.

In looking at the code, however, you have no idea of what's going on in the code unless you understand JavaScript. Sometimes the generated code will tap into code in one of its libraries that you cannot see. It will connect to an external .js file containing a good deal of code that you won't see, but can be fairly complex. If you want to change it or even tweak it a little with added or altered code, you're lost. So, the first goal in learning JavaScript is to be able to fine-tune JavaScript generated in your web site development tools.

Second, you might want to learn JavaScript to lighten the amount of code that some site development tools generate. Jakob Nielsen, author of *Designing Web Usability* (New Riders, 1999), points out that site-development tools sometimes cause "code bloat." That is, the generic code developed in the site tools sometimes generates more code than is absolutely necessary, and you can cut it down. The reason you want to avoid code bloat is that it creates a larger file than you need, so it loads slower. For example, take a look at the following script:

```
<html>
<head>
<title>Swap Simple</title>
<script language="JavaScript">
var sample2= new Image( )
sample2.src="sample2.gif"
var sample1=new Image( )
sample1.src="sample1.gif"
function switch2( ) {
            document.sample.src=sample2.src
}
function switch1( ) {
            document.sample.src=sample1.src
}
</script></head>
<body>
<a href="#" onMouseOver="switch2( )" onMouseOut="switch1( )"> <img
name="sample"
src="sample1.gif" border=0></a>
</body>
</html>
```

This code is quite simple (if you know JavaScript) and accomplishes the same rollover effect as code generated in one of the web site development tools. However, it is a good deal clearer to understand because all of the elements are laid out and available, whereas code generated by site development tools can often mask many of the key elements that make your code do what you want.

Summary

This chapter's goal has been to provide a glimpse of JavaScript and a little understanding of what it can do. JavaScript ranges from very simple but useful applications to very complex scripts that handle complex tasks. Don't expect to learn the language all at once, but do begin to think about designs that can be aided by the inclusion of JavaScript in your web pages. Include interactive elements in your pages using JavaScript so that the viewer is engaged in the page rather than a passive audience. Like all languages, the more you use it, the easier it becomes to apply in your web pages. So, start using the language right away.

JavaScript is interpreted by your browser, and while different browsers can require different scripting strategies, most JavaScript follows the ECMA-262 standard. However, be aware that some nagging differences exist. In later chapters, you will be prepared with strategies for dealing with these differences.

Chapter 2. An Orientation to JavaScript

CONTENTS>>

Writing JavaScript

AS YOU SAW IN Chapter 1, “Jump-Starting JavaScript,” JavaScript goes into an HTML page. However, you do not write JavaScript with the same abandon as you do HTML. Very specific and apparently minor differences exist between how HTML can be written and how JavaScript can be written. While the differences might appear to be minor or even trivial, if the rules for writing JavaScript are not followed, you can run into glitches. This chapter examines the nuances of JavaScript so that when you start writing your own scripts, you’ll have all of the basics clear in your mind.

HTML is a markup language, and JavaScript is a programming or scripting language. HTML describes what is to be presented on a page, and JavaScript dynamically changes what is on an HTML page (among other tasks.) Both use code. HTML’s code is in a series of angle brackets that describe how to treat the material between the opening and closing brackets. JavaScript is a set of statements and functions that does something in an HTML page. JavaScript can refer to and alter objects described by HTML.

Case Sensitivity

You can write HTML tags in just about any way you want, as long as you spell the tags correctly and remember to include the arrow bracket around the tags. For example, the following little page will work just fine in HTML:

```
<hTmL>
<heaD>
<Title>Do it your way</tITLE>
</HEAD>
<BoDY Bgcolor="HotPink">
<ceNTer>
<h1>
HTML is CaSe InSeNsItIvE!
</H1>
```

```
</bODY>
</HTML>
```

Just about every non–case-sensitive combination of characters that you can imagine has been put into that page. The opening tags of a container are in one case combination, and the closing tags are in another. Tags in one case are duplicated with tags in another case. For HTML, that's no problem. You don't have to pay attention to case at all.

JavaScript is the opposite. You have to pay attention to the cases of everything that you type in JavaScript because it is case-sensitive. The HTML around the script need not be case-sensitive, but the JavaScript itself must be. Consider the following examples. The first follows the rules of case sensitivity, and the second one does not.

```
<html>
<head>
<title>Case Sensitive</title>
<script language="JavaScript">
alert("Pay attention to your cases!");
</script>
</head>
<body bgcolor="moccasin">
<p>
<h1>Just in case!</h1>
</p>
</body>
</html>
```

When you load the page, you will see an alert message telling you to pay attention to your cases. As soon as you click the OK button on the alert box, the rest of the page appears with the message "Just in case." Now, look at this next script to see if you can tell where the error lies. It is slightly different from the first—only the *a* in "alert" has been changed so that it is "Alert." Just that little change will invalidate the JavaScript code. Launch the page with the capital *A,* and see what happens.

```
<html>
<head>
<title>Case Sensitive</title>
<script language="JavaScript">
Alert("Pay attention to your cases!");
</script>
</head>
<body bgcolor="moccasin">
<p>
<h1>Just in case!</h1>
</p>
</body>
</html>
```

As you saw, the page didn't crash and burn. It just ignored the JavaScript and went on and put up the page on the screen. I would rather see an error message issued so that I could see any problems that arise, but neither of the newest versions of the browsers indicated any trouble at all. (In Netscape Navigator 4.7 , a little error message blinks in the lower-left corner, but it happens so fast that you cannot tell that your script has an error.) Debugging JavaScript is often a matter of *not seeing* what you expect on the screen rather than seeing any clue that you've coded your script incorrectly. However, ignoring case sensitivity is likely to be one bug in the code that you should suspect immediately.

For the most part, JavaScript is typed in lowercase fonts, but you will find many exceptions to that rule. In Chapter 1, you might have noticed the use of `Math.floor` along with `toString` in one of the scripts. Both of those words use a combination of upper- and lowercase fonts: intercase. `Object`, `Math`, `Date`, `Number`, and `RegExp` are among the objects that use case combinations as well. Properties such as `innerHeight`, `outerWidth`, and `isFinite`, likewise, are among the many other JavaScript terms using case combinations.

You also might run into cases differences in HTML and JavaScript. Event-related attributes in HTML such as `onMouseOver`, `onMouseOut`, and `onClick` are spelled with a combination of upper- and lowercase characters by convention, but, in JavaScript, you must use all lowercase on those same terms. Hence, you will see `.onmouseover`, `.onmouseout`, and `.onclick`.

Another area of case sensitivity in JavaScript can be found in naming variables and functions. You can use any combination of upper- and lowercase characters that you want in a function or variable name, as long as it begins with an ASCII letter, dollar sign, or underscore. Functions are names that you give to a set of other statements or commands. (See an introduction to functions in Chapter 1.) Variables are names that you give to containers that hold different values. For example, the variable `customers` might contain the words "John Davis" or "Sally Smith." Variables can contain words or numbers. (See the next chapter for a more detailed discussion of variables.) When the function or variable is given a name, you must use the same set of upper- and lowercase characters that you did when you declared the variable or functions. For example, the following script uses a combination of characters in both variables and function. When the function is fired, the name must be spelled as it is in the definition.

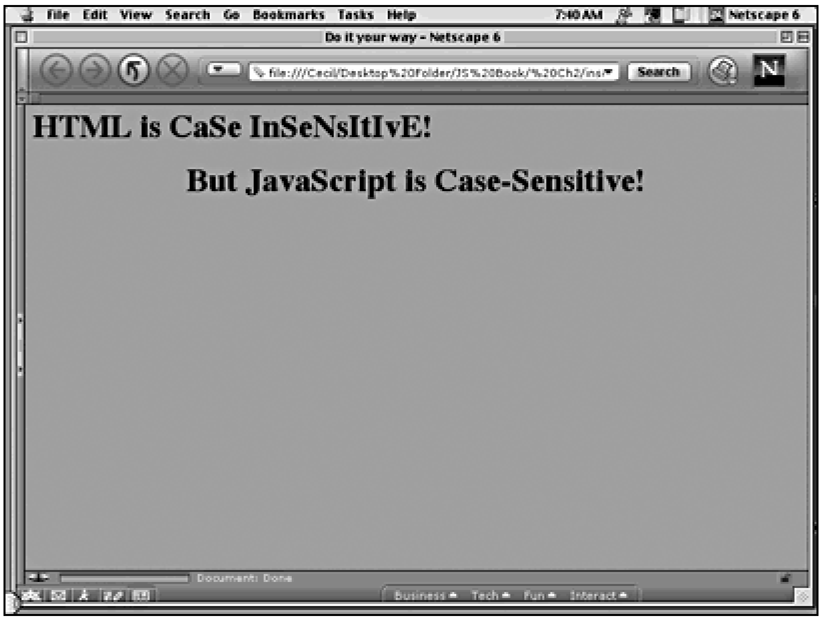
VarFuncCase.html

```
<html>
<head>
<title>Cases in Variables and Functions</title>
<script language="JavaScript">
var Tax=.05
function addTax(item) {
var Total=item + (item * Tax);
var NewTotal=Math.floor(Total);
var Fraction=Math.round(Total *100)%100;
if (Fraction<10) {
Fraction = "0" + Fraction;
}
Total=NewTotal +"." + Fraction;
alert("Your total is $" + Total);
}
</script>
</head>
<body bgcolor="palegreen">
<center><h2>
<a href=# onClick="addTax(7.22)";> Click for total </a>
</body>
</html>
```

Several variables and two functions (the `alert` function is built in and so has a name already—alert) are included in the script, but notice that all of the variable names and function references use the same combination of upper- and lowercase characters. The string message for the `alert` function reads, `"Your total is $" +Total);`. The first use of `total` is part of a message (string literal) and is not a variable in this case. The `Total` attached to the end of the alert message, however, is a variable, and it uses the uppercase first letter as the variable does when it is defined. Likewise, the argument in the function (`item`) is always referenced in lowercase because it is initially written in lowercase. The variable declaration lines beginning with `var` signal the initial creation of a variable, and the case configuration used in those lines is the configuration that must be used throughout the page in reference to a given variable. Chapter 3, "Dealing with Data and Variables," explains developing variables in detail.

Figure 2.1 shows what you will see when the page loads and you click the link text. If you want a link to launch a JavaScript function, you can use a "dummy" link by inserting a pound sign (`#`) where the URL usually is placed. Then, by adding an event handler, you can launch the function. Try changing the value in the `addTax( )` function to see what you get. Also, see what happens when you change `addTax` to `ADDTAX( )`.

Figure 2.1. None of HTML is case-sensitive, but virtually all of JavaScript is.



File Edit View Search Go Bookmarks Tasks Help
7:40 AM
Netscape 6
Do it your way - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/%20Ch2/ins
Search
Document: Done
Business Tech Fun Interact

HTML is CaSe InSeNsItIvE!

But JavaScript is Case-Sensitive!

Entering Comments

Comments are messages to yourself and others who are working to develop a JavaScript program with you. They serve to let you know what the following lines of code do or, if incomplete, what you want them to do. Comments in JavaScript are entered by prefacing a line with double forward slashes (`//`). When the code is parsed in the browser, all of the lines beginning with the double slashes are ignored. For example, the following code segment shows a reminder to add tax to an item in an e-business application:

```
//Include a variable to add taxes
```

```
tax= .06
//Add the tax to the taxable item
item += item * tax
```

The bigger and more complex your scripts become, the more you will need to have well-commented code. Comments in code become even more crucial when you are working with a team to create a web site and others need to know what your code is doing. In this book, you will see comments throughout the code in the larger scripts to point out different elements. In shorter scripts, the comments are in the text of the book, a luxury that you will not have in your own coding. Remember to comment your code, and you will see that you can save a good deal of time reinventing a solution that is already completed.

The Optional Semicolon

Several languages that look a lot like JavaScript require a semicolon after lines. For example, Flash ActionScript and PHP (see Chapter 14, "Using PHP with JavaScript," and Chapter 18, "Flash ActionScript and JavaScript") both require semicolons. Likewise, compiled languages such as C++ and Java require semicolons at the end of lines. JavaScript made semicolons optional.

So, the question is, do you really need the semicolon? JavaScript places "invisible" semicolons at the end of each line, and by placing a *visible semicolon,* you can better see what's going on. For debugging your program, the semicolons alert you to where a line ends; if you did not intend a line to end where you put a semicolon, you can better see the error. Hence, the answer to the question of whether you should include semicolons is "yes."

Semicolons go at the end of lines that do not end in a curly brace or to separate variable declarations on the same line. For example, the following two code segments show where semicolons may optionally be placed.

```
function findIt( ) {
      if(x="searchWord") {
            document.formA.elementA.value=x;
      }
}
```

Because four of the five lines end in a curly brace, only the third line optionally can have a semicolon. On the other hand, in a list of variable definitions, you can place a semicolon at the end of every line.

```
var alpha="Apples";
var beta= alpha + "Oranges";
var gamma= Math.sqrt(omega);
var delta= 200/gamma;
```

Older Browsers

At the time of this writing, *Netscape Navigator 6.01* is in general release for both Windows and Macintosh operating systems, and Internet Explorer has a Version 6 in public preview for Windows and is in Version 5.5 on the Macintosh. By the time this book is published, both major browsers will most likely have Version 6 as their standard browser. Keeping in mind that the browsers are the interpreters for

JavaScript, the version of browser that others use to view your scripts is very important. Version 3 browsers will read most JavaScript, but not until Version 4 of the two major browsers was JavaScript 1.2 available. Therefore, you really need your viewers to have at least Version 4 of either major browser for use with code from JavaScript 1.2. A guy in Outer Mongolia with an Internet connection has the same access to a new browser as a guy in Silicon Valley; all he has to do is to download and install either browser for free.

However, to get around the holdout who thinks that technology ended with his Version 2 Netscape Navigator, you can enter a simple set of semitags to mask the JavaScript. Because the older browsers don't know JavaScript from Sanskrit, they think that the code is text to be displayed on the page. To hide the JavaScript, you can place the container made up of `<!-`and `//-->` around the JavaScript code. For example, the following script segment is hidden from older browsers, and their parsers will skip over it:

```
<script language="JavaScript">
<!-
document.write("The old browsers cannot read this.")
//-->
```

Rarely do you find anyone still using browsers older than Version 4, and unless you want to degrade your JavaScript to an earlier version, you can include the masking container. However, at this point in browser development, it might be wiser to let the visitor know that her browser could use an upgrade by allowing JavaScript to appear on the screen. (A few cantankerous designers even use notes telling the viewer to upgrade his browser or get lost!)

Some designers attempt to write JavaScript with different sets of code for users with very old browsers by using browser-detection scripts written in JavaScript. In that way, users with older browsers can see some web page. In terms of cost benefits, having alternative sets of code for different browsers, different versions of browsers, and browsers for different platforms can become an onerous and expensive task. However, each designer/developer needs to decide her own willingness to have several different scripts for each page. With short scripts and a few pages, only a little extra work is required. However, with big sites and long sets of code, designers might find that they have to increase their time on the project and that they must charge their clients, or use the lowest common denominator of JavaScript.

Naming Rules and Conventions

All the rules for naming functions and variables discussed previously in the section on case sensitivity apply. If you name your variables or functions beginning with any letter in either uppercase or lowercase or with an underscore or dollar sign, the name is considered legitimate. For the most part, in naming variables, avoid using dollar signs as the first character because they can be confusing to Basic programmers, who might associate the dollar sign with a string variable, or PHP programmers, who begin all variables with a dollar sign.

You cannot have spaces in the names that you use for variables or functions. Many programmers use uppercase letters or underscores to make two words without a space. For example, the following are legitimate variable names where a space might be used in standard writing:

```
Bill_Sanders= "JavaScript guy";
BillSanders= "JavaScript guy";
Free$money="www.congame.html";
```

Whether to use an underscore or a cap to begin a new word in a variable or function name is your own preference.

Reserved Words

Reserved words are those in JavaScript that have been reserved for statements and built-in functions. If you use a reserved word for a variable or function name, JavaScript might execute the statement or function for the reserved word and not your variable or function. For example, if you use the reserved `break` for a variable name, JavaScript might attempt to jump out of a loop. Table 2.1 shows a list of current and future JavaScript reserved words.

Table 2.1. Reserved Words

abstract	final	public
boolean	finally	return
break	float	short
byte	for	static
case	function	super
catch	goto	switch
char	if	synchronized
class	implements	this
const	import	throw
continue	in	throws
debugger	instanceof	transient
default	int	true
delete	interface	try
do	long	typeof
double	native	var
else	new	void
enum	null	volatile
export	package	while
extends	private	with
false	protected	

You should also try to avoid words that you will find used in statements, methods, and attributes. For example, `name` and `value` are attributes of forms and should be avoided. As you learn more JavaScript, try to avoid the new words that you learn as names for variables and functions.

A Weakly Typed Language Means That JavaScript Is Smart

One last characteristic of JavaScript needs to be discussed before going on to the next chapter. JavaScript is considered a "weakly typed" or "untyped" language. The *type* in question here are the *data types*—nothing to do with typing from your keyboard. For programmers coming from C++ or Java, two strongly typed languages, this means that JavaScript will figure out what type of data you have and make the necessary adjustments so that you don't have to redefine your different types of data. Designers new to programming will welcome a weakly typed language because it will save time in learning several different conversion steps and data type declarations.

For example, suppose that you're setting up an e-business site with lots of financial figuring that you want to slap a dollar sign on when all of the calculations are finished. Performing calculations involving money requires floating-point numbers. However, as soon as a dollar sign is added to the number, it becomes a string variable. In strongly typed languages, you need to convert the floating-point number to a string and then concatenate it with the dollar sign. Consider the following example using JavaScript to add five different items and placing the whole thing into a string.

WeaklyType.html

```
<html>
<head>
<script language="JavaScript">
var apples=1.43;
var oranges=2.33;
var pears=4.32;
var tax=.04;
var shipping=2.75;
var subtotal=apples + oranges + pears;
var total=subtotal + (subtotal * tax) + shipping;
var message="Your total is $";
var deliver= message + total +".";
document.write(deliver);
</script>
</head>
<body bgcolor="indianred">
</body>
</html>
```

Of course, when the calculations are all complete, you could drop the decimal points to two, as was done in the example script `VarFuncCase.html` in the previous section. However, the point is that no type definitions were required. Strings and numbers were merrily mixed with no cause for concern. Although the term "weakly typed" might imply some deficiency with JavaScript, the term actually means that JavaScript is smart enough to do the work of determining what type of data any given variable should be. In learning about data types and variables in the next chapter, you will be very grateful that JavaScript (and not you) does most of the work in figuring out data types.

Summary

The purpose of this short chapter has been to get you off on the right foot when you begin writing your own scripts. The most frustrating experience in learning how to program a new language is debugging it when the outcome is not the one expected. By paying attention to certain details and learning a set of rules for writing JavaScript, not only are you less likely to run into bugs when you create a page that includes JavaScript, but you are more likely to locate and correct them when they do pop up.

Chapter 3. Dealing with Data and Variables

CONTENTS>>

While the heart of a scripting or programming language is contained in its statements and structure, its utility is in the way the language handles data. As noted in Chapter 2, “An Orientation to JavaScript,” JavaScript is a weakly typed or untyped language, so the burden of keeping track of data types is left largely to the inner workings of the language.

However, JavaScript is more sophisticated than you might expect of a weakly typed language. In addition to strings, numbers, and Boolean values, JavaScript data can include objects, including arrays and functions. Each of these data types is important to understand before proceeding so that when you’re working on your design, you can decide which data types are required to create a certain effect. If you’re new to programming, you might need to study the data types carefully; while a newcomer to programming need not fully understand all the nuances of the data types immediately, trying out different types of data and experimenting with their characteristics is important. Later, as you gain experience, you will be able to solve many design problems with a clear understanding of the types of data that JavaScript generates and you’ll be solving programming problems that help execute your design.

Literals

The raw data that make up the root of data types are called “literals.” These are, in effect, *literally* what they represent themselves to be. Numbers, strings, and Boolean values make up the core set of literals in JavaScript. Little mystery exists with literals, but important differences exist between them.

Numbers

The fundamental data in most computer languages are numbers. Because JavaScript is weakly typed, all numbers are treated as floating-point, so you need not distinguish between integers and floating-point literals. All of the following values are treated as numeric literals:

```
223.48
20
0
```

```
500.33
```

When assigning numeric literals to names (identifiers), you simply write them in their raw form, with no required quotation marks or other characters, to represent decimal values. With numbers other than decimal values or very large numbers, special requirements exist.

Scientific Notations

If you need scientific notations or are returned a value written in a scientific notation, you will be glad to know that they are written in standard format. For example, you might see the value `9.00210066295925e+21` on the screen after your script has calculated some really big numbers.

The letter `e` is followed by a plus or minus sign and from one to three integers. (The sign is placed in a returned result but is optional if you write your own notation.) The integers following the `e` are the exponent, and the rest of the number (preceding the `e` notation) is multiplied by 10 to the power of the exponent. In most JavaScript applications, numbers with scientific notations do not appear, but if they do, they are treated for purposes of calculations just like any other number.

Hexadecimal Literals

Base 16 or hexadecimal literals have a special preface to alert the parser that the combination of numbers and letters is indeed made up of special values. All hexadecimal literals are prefaced by `0x` (zero-x), followed by 0–9, A–F characters indicating a hexadecimal value. For example, the color red in hexadecimal is `FF0000`; in JavaScript, it is written as `0xFF0000`.

All calculations done in hexadecimal values in JavaScript are returned as decimal values. For example, if you add `0xa8` to `0xE3`, the resulting value in decimal is `395` instead of the hexadecimal value `18b`. Fortunately, JavaScript provides a way to express hexadecimal values using the `toString( )` method. (The decimal-tohexadecimal conversion script in Chapter 1, "Jump-Starting JavaScript," used the same method.) By including the number base as an argument, you can return a hexadecimal value. The following script shows how.

hexNota.html

```
<html>
<head>
<title> Hexadecimal Values </title>
<script language="JavaScript">
var alpha=0xdead;
var beta=0xbeef;
var gamma=(alpha + beta).toString(16);
document.write(gamma);
</script>
</head>
<body bgcolor="springgreen">
</body>
</html>
```

Hexadecimal values' most familiar application in JavaScript and HTML is as sixcharacter color values. It requires no calculations for a color other than conversion from decimal. However, many other occasions might arise in which calculations using hexadecimal values occur, and knowing how to generate hexadecimal results in JavaScript can be useful.

Strings

Like other programming languages, string literals are text. Any set of characters placed in quotation marks (single or double) make up a string literal. For example, the following are all string literals:

```
"JavaScript makes things jump."
'1-2 Buckle your shoe'
"54321"
"200 Bloomfield Avenue"
'My dog is named "Fred"'
```

Numbers in a string are treated as text, not as values that can be calculated. Quotation marks within quotation marks need to be nested. That is, if the initial quotation is a double quote, the two single quote marks must be within the double quotes, or vice versa, as in the last example. For example, the output of the string literal `‘My dog is named “Fred”’` returns

```
My dog is named "Fred"
```

The Escape Sequence for Strings

You may also include escape characters and sequences by prefacing a code with a backslash (`\`) for additional control over string literals. For example, the literal `\’` prints an apostrophe without affecting the literal itself. Other escape codes include the following:

- \n New line
- **\’** Single quote or apostrophe
- **\”** Double quote
- \\ Backslash

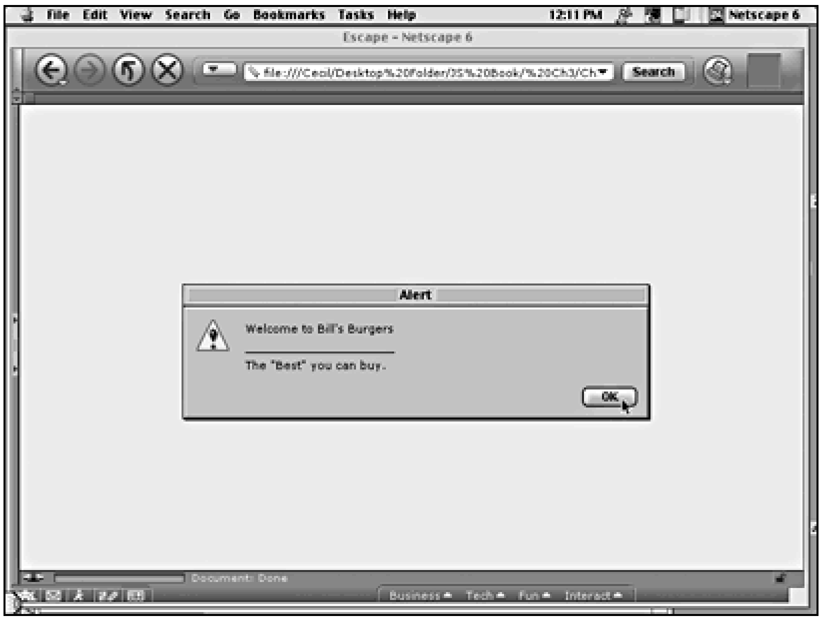
The following script shows how you can use strings, including escape sequences, to order the output in an `alert( )` function.

escape.html

```
<html>
<head>
<title> Escape </title>
<script language="JavaScript">
var alpha="Welcome to Bill\'s Burgers\n";
var beta="______________________";
var gamma="\nThe \"Best\" you can buy."
alert(alpha + beta + gamma);
</script>
</head>
<body bgcolor="palegoldenrod">
</body>
</html>
```

Figure 3.1 shows the outcome that you can expect when you launch the program.

Figure 3.1. JavaScript uses escape sequences to format data.


File Edit View Search Go Bookmarks Tasks Help
12:11 PM
Netscape 6
Escape - Netscape 6
file:///Ceal/Desktop%20Folder/JS%20Book/%20Ch3/Ch
Search
Alert
OK
Document: Done
Business Tech Fun Interact

Welcome to Bill's Burgers

The "Best" you can buy.

While the formatting using the escape sequences works well with the `alert( )` function, it does not work the same with `document.write( )`. The character substitutions for apostrophes and quotes return the same results, but the `\n` (new line) sequence does not. Because `document.write( )` places text into the HTML page itself rather than a special box, as do the `alert( )` and `prompt( )` functions, you can use HTML tags such as `<br>` to achieve a new line with `document.write( )`. However, if you attempt to use HTML formatting tags with the `alert( )` or `prompt( )` functions, you will find that they either will not work as expected or will not work at all.

Boolean Values

A Cork University professor named George Boole developed a mathematical system of logic that became known as *Boolean mathematics.* Much of the logical structure of computer engineering and programming is based on Boole's system. The Boolean values in JavaScript are two literals, `true` and `false` (`1` or `0`, `yes` or `no`). The Boolean literals are derived from logical comparisons, testing a truth value and then using that value (`true` or `false`) for another operation. Most common Boolean tests and values are found in conditional statements that have this form:

```
If a condition is true
       Flag=true
Else
       Flag=false
```

Using the flag value, different paths are followed, usually in the form of a statement. Chapter 5, "JavaScript Structures," covers conditional statements in detail; in that chapter, you will see extensive Boolean literals used.

Another way that a Boolean literal is generated and used is with comparison operators. For example, the following sequence results in a Boolean literal of `false`, `0`, or `no` for the `c` variable. (The `>` operator represents "greater than.")

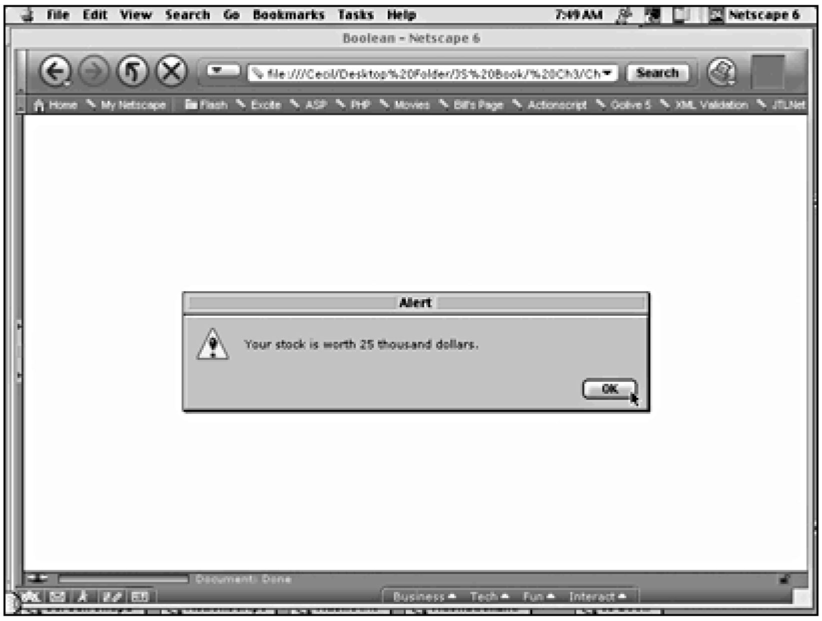
```
var a=10;
var b=20;
var c=(a > b);
```

The Boolean literal is automatically changed by JavaScript to a `true/false`, `1/0`, or `yes/no` value, depending on the context. For example, the following little script uses the Boolean literal to generate a value much larger than `1`:

```
<html>
<head>
<title> Boolean </title>
<script language="JavaScript">
var a=5;
var b=6;
var c=(a<b)
var d="Your stock is worth " +( c * 25 )+" thousand dollars.";
alert(d);
</script>
</head>
<body bgcolor="blanchedalmond">
</body>
</html>
```

Figure 3.2 shows the calculated output in JavaScript.

Figure 3.2. The output in the alert box has been formatted with the calculated results of the JavaScript program.



Your stock is worth 25 thousand dollars.

Whenever a result is `true` (`1` or `yes`), JavaScript automatically looks at the context of the literal's use and makes the appropriate changes. Simply multiplying by 1 or 0 makes a huge difference because multiplying by 0 always results in 0. Sometimes, using tricks with Boolean values, you can save steps in your programs and do some interesting things that you would not be able to do otherwise.

Calculations and Concatenations

JavaScript numbers and strings sometimes share common operators, and others are used with numbers only. Chapter 4, "Using Operators and Expressions," covers all the operators and how they are used, but the basic arithmetic operators of addition, subtraction, multiplication, and division are `+`, `-`, `*`, and `/`, respectively. The plus sign (`+`) is used both for adding numbers and concatenating strings. String concatenation refers to binding one or more strings into a single string. For example, this returns `New Riders`:

```
var firstName="New";
var lastName="Riders";
var gap=" ";
var publisher=firstName + gap + lastName;
document.write(publisher);
```

The plus sign (`+`) joins the three string variables `firstName`, `lastName`, and `gap`.

In addition to the basic math operators, JavaScript has both `Math` and `Number` objects for more complex operations and constants. Chapter 7, "Objects and Object Hierarchies," deals with these objects, but you should know of their availability for special number cases and complex math. The `Math` object has been employed in some of the examples where a conversion was required. For example, `Math.floor( )` was used in Chapter 1 to convert the value of a form window into a number. All data entered into a text window *are* treated as text, so by using the `Math.floor( )` function, it was possible to both round any value down to the nearest integer *and* convert it to a number. Elsewhere in this chapter, the `toString( )` function changed a decimal value into a hexadecimal string representation.

Two important built-in functions may also be used for string and number conversions, `parseFloat( )` and `parseInt( )`. The following script shows how these two different functions work.

parseString.html

```
<html>
<head>
<title> Parse them Strings! </title>
<script language="JavaScript">
var elStringo="14.95";
var laStringa="32 on sale";
var newVal=parseFloat(elStringo);
var hotVal=parseInt(laStringa);
document.write("Your total is: $" + (newVal+hotVal));
</script>
</head>
<body bgcolor="gainsboro">
</body>
</html>
```

Two strings containing numbers, `elStringo` and `laStringa`, are converted into numbers. The `parseFloat( )` method takes both numbers and the first decimal and numbers beyond the first decimal. When it encounters a non-number after the first decimal point, it ignores characters and the function transforms the values to the left into a floating-point number. The `parseInt( )` method parses numbers until the first non-number is encountered. Everything to the left of the first non-number is converted into a number. (Because all numbers in JavaScript are essentially floating-point numbers, the "integer" is simply a floating-point value with no decimal value other than zero.)

Objects as Literals

Objects are a collection of properties. While objects are discussed in detail in Chapter 7, you should understand that they can be treated as literals and placed into variables. In designing a dynamic web site, you can create the objects that you need in JavaScript and use them repeatedly on different pages. Think of objects as "design clusters" that can be placed where you need them.

As a collection of properties, objects can be used to create very useful tools and serve as the basis of object-oriented programming. To create an object, you first give it a name as a new object:

```
var shopCart = new Object();
```

When you have established your object, you can begin adding properties to it:

```
var shopCart.item = 5.95;
var shopCart.tax = .06;
var shopCart.shipping = 14.95;
```

When making object literals, each property can be defined within curly braces, with the properties separated by commas. Each property is separated from its value by a colon.

The following shows a simple example of how to work with object literals:

```
var shopCart = { item: 5.95, tax: .07 , shipping: 14.95 };
```

The outcome is identical to the first set of object properties defined previously, but the format is different. Object literals, along with other characteristics and uses of objects, are revisited in Chapter 7 in more detail. For now, it is enough to understand that object literals constitute one of the data types in JavaScript.

Functions as Literals

Chapter 6, "Building and Calling Functions," covers functions in detail. In this section, the focus is on functions as a data type. Functions can be built-in or user-defined. A user-defined function has this form:

```
Function functionName(optional argument) {
Commands, statements, definitions
} //function terminator
```

Functions can be launched in different places in a program and serve as self-contained groupings of code to accomplish one or more tasks in a program.

However, JavaScript has the capability to include a function in a variable as a literal. For example, using the Base 16 conversion with the `toString( )` method, it's possible to create a literal that converts decimal to hexadecimal. Base 16 is the name used for hexadecimal values. The decimal system is Base 10 because the number system is based on 10 characters, 0–9. Base 16 has 16 characters, 0–F. The following script changes three decimal values into a single hexadecimal value and slaps a `#` sign on the front to remind you that it's a hexadecimal value. The function named `converter` that does the conversion becomes a literal in a variable definition. The `converter` function runs three times to generate color values for the `R`, `G`, and `B` values.

functionLit.html

```
<html>
<head>
<title> Function Literal </title>
<script language="JavaScript">
function converter(decNum) {
      if (decNum >=16) {
            return decNum.toString(16);
      } else {
            return "0" + decNum.toString(16);
      }
}
var colorIt="#" + (converter(255) + converter(108) + converter(4));
document.write(colorIt);
</script>
</head>
<body bgcolor="ghostwhite">
</body>
</html>Objects
```

You can never have too many decimal-to-hexadecimal conversion scripts. Compare the previous conversion script with the one in Chapter 1. All the decimal input for the one in this chapter is in the script, and all the data use functions as literals.

Undefined and Null Values

Two other types of data, `null` and `undefined`, should help you better understand a little about how JavaScript deals with variables. Both generally signal something that you forgot to do, but sometimes they are a planned part of a script.

An `undefined` value is returned when you attempt to use a variable that has not been defined or one that is declared but that you forgot to provide with a value. A nonexistent property of an object also returns `undefined` if it is addressed.

On the other hand, `null` amounts to a "nothing literal." You can declare and define a variable as `null` if you want absolutely nothing in it but you don't want it to be undefined. `Null` *is not* the same as zero (0) in JavaScript. In some situations, you will want to find the result of a calculation, which could be zero;

rather than putting in a zero to begin with, you can place a `null` in a variable. In that way, when data does come into a variable, you will know that the value is part of the calculation and not a default zero that you placed there. The following little script shows the different outcomes when a variable is undefined and when it is null.

nullUndef.html

```
<html>
<head>
<title> Null and Undefined </title>
<script language="JavaScript">
var nada;
var noVal=null;
document.write("<b>No value assigned and the variable is " + nada +
".<p>" );
document.write("A null value was assigned and returns " + noVal +
".</b>");
</script>
</head>
<body bgcolor=#BadCab>
</body>
</html>
```

Regular Expression Literals

Regular expression (RE) literals are advanced programming concepts and constitute almost a whole new language. In fact, learning the regular expression nomenclature will prepare you for much of Perl. (Chapter 16, " CGI and Perl," covers working with JavaScript, CGI, and Perl.) The easy part of regular expression literals is that they are identified by slashes (`/`), as strings are with quote marks. The following are some examples of RE literals:

```
/do/
/[a-k]/
/[^l-z]/
```

The meaning of the RE literals is another matter. Typically, RE literals are used with sequences or groups. Groups are placed in brackets (`[]`), and sequences are indicated by a dash (`-`). For example, a group of letters that you might want to match might be `l`, `k`, and `r`. Any one of these letters would be recognized if placed into a class `/[lkr]/`, while a word such as *fun* would be patterned as `/fun/` in an RE literal. (A class in regular expressions refers to a group of characters to match.) A range of characters or digits would be placed in a pattern such as `/[b-s]/` or `/[5-9]/`. Anything that you don't want is prefaced by a caret (`^`), as in `/^blink/`.

For the designer, regular expressions are important for searching for different elements of strings. If your design requires finding keywords entered by customers looking for certain products, you could use regular expressions to search for the product and provide feedback to the customer about whether it is available.

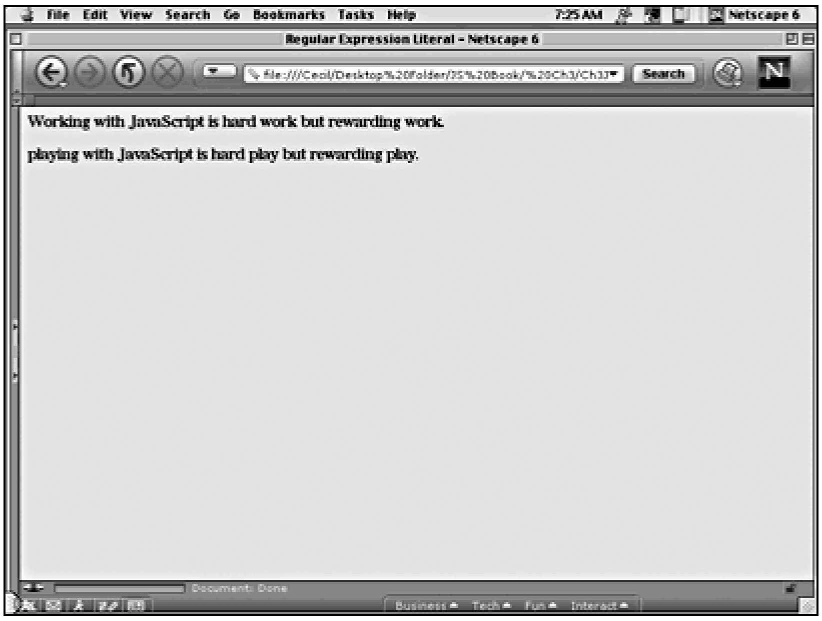
An example of an RE literal can be seen in a global replacement of one word for another in the following script.

regExp.html

```
<html>
<head>
<title> Regular Expression Literal </title>
<script language="JavaScript">
var work="Working with JavaScript is hard work but rewarding work.";
var play=work.replace(/work/gi, "play"); //Regular expression
document.write("<p><b>" + work + "<p>" + play);</script>
</head>
<body bgcolor="navajowhite">
</body>
</html>
```

The RE literal `work.replace(/work/gi, "play");` commanded the variable named `work` to have all instances of the word *work* (`/ work/`) be globally (`g`) replaced, ignoring case (`i`), by the word *play.* The output for the script is shown in Figure 3.3.

Figure 3.3. Using regular expressions in variables, parts of a string can be changed.



Working with JavaScript is hard work but rewarding work.

playing with JavaScript is hard play but rewarding play.

Because cases are being ignored, "Working" becomes "playing." Otherwise, the variable containing the RE literal simply took the first string and replaced it with the second following the rules of regular expressions.

In Chapter 16, you will revisit regular expressions in Perl and see how JavaScript can use regular expressions themselves in dealing with data in a CGI bin.

Variables

If you have been following the book sequentially, by this point, you probably have a pretty good idea what a variable is from all of the examples. You've been provided with several of the rules for providing names and values for variables as well. This section examines variables in detail and spells out what you can do with them and how to use them.

I like to think of variables as containers on a container ship. You can put all different types of content into the containers, move them to another port, empty them, and then replace the container with new content. However, the container ship analogy suffers when you realize that the content in the containers must have magical properties. If you have a container full of numbers and you add a string, the whole container magically becomes a string. Because JavaScript is untyped (or weakly typed), both the contents and the characteristics of the variable can change. The main point, though, for readers new to the concept of a variable is that variables are containers with changeable contents.

Declaring and Naming

JavaScript, like most scripting languages, has two basic ways of declaring a variable. First, as you have seen throughout the book up to this point, variables are declared using the `var` word. You simply type in `var` followed by a variable name and value. The following are typical examples:

1. **`var item;`**
2. **`var price= 33.44;`**
3. **`var wholeThing= 86.45 + (20 *7);`**
4. **`var name="Willie B. Goode";`**
5. **`var address "123 Elm Street";`**
6. **`var subTotal=sumItems`**
7. **`var mixString= 11.86 + "Toy Cats";`**
8. **`var test = (alpha > beta)`**

By taking each variable one at a time, you can see how the different data types discussed are placed into a variable:

1. The first example demonstrates that you can declare a variable but not give it a value. As you saw previously in this chapter, such variables have an *undefined* value.
2. The second variable contains simple *primitive* data—a numeric literal with a value of `33.44`.
3. The third variable is a compound variable made up of a numeric primitive and a compound expression.
4. The fourth variable is defined as a simple string literal.
5. The fifth variable is also a simple string literal, but it uses a mix of digits and letters.
6. Sixth, the variable is defined with another variable.
7. The seventh variable is a mix of a numeric primitive and a string primitive, creating a string variable.
8. Finally, the last variable is a Boolean value derived from compound data.

Declaring a variable alerts the computer to the fact that a new variable is available to use. After a variable is declared, it need not be declared again. For example, in loop structures, the `counter` variable can be defined in the initialize section, but not in the test or change (increment/decrement) sections. For example, the following code segment shows that the variable named `counter` is declared in the first segment but then is not declared again:

```
for (var counter=0; counter < 40; counter++) {....
```

Some programmers like to initialize all of their variables at the beginning of a script with undefined values. Then later they can use them without having to remember to add `var`. Also, you can have a single line with several variable definitions, with each variable separated by a comma *or* a semicolon, as the following script illustrates.

clutter.html

```
<html>
<head>
<script language="JavaScript">
var a=20; b=30, c="wacka wacka do"; gap=" ";
document.write(a+ gap + b + gap +c);
</script>
<body bgcolor=#C0FFEE>
</body>
</html>
```

I generally avoid declaring more than a single variable on a line. Multiple declarations in a line, while workable, can clutter what has been defined and what a variable has been defined as. The script clutter.html amply illustrates such confusion. (By the way, the character following the `c` in the `bgcolor` value is a zero, [0], not a capital *O.*)

You may omit the `var` keyword in your variable declarations, and you undoubtedly will see scripts in which the programmers have done so. For example, the following are perfectly good examples of such declarations:

```
acme = "The Best";
cost = 23.22;
```

While JavaScript accepts these declarations for global variables, you can run into problems elsewhere by omitting `var`. (See the note in the following section.) Thus, for a good programming habit that will avoid problems, *always* use the `var` keyword when declaring a variable.

Global and Local Variables

Variables in JavaScript have *scope.* The scope refers to the regions of the script where the variables can be used. A global variable, as the name implies, has global scope and is defined in the entire script. Local variables are local to the functions in which they are defined. As a *general* rule, avoid naming any two variables, whether local or global, with the same name or identifier.

NOTE

While using the keyword `var` *is optional in declaring global variables, problems can arise if you do not incorporate* `var` *in defining your local variables. When using* `var` *in a local variable declaration, the program recognizes it* as a local variable, *not a change in the value of a global variable. With no* `var` *keyword used, your script cannot tell the difference, and you risk inadvertently changing the value of a global variable. The moral to this story is to always use the* `var` *keyword for variable declaration.*

Within a function, a local variable has precedence over a global variable of the same name. So, if your global variable named `ID` has the value `Fred`, and a function also with a variable named `ID` has a value of `Ethel`, the name `Ethel` will appear when the function displays the variable's value. However, if you display the value of the `ID` variable from outside the function, the value will be `Fred`.

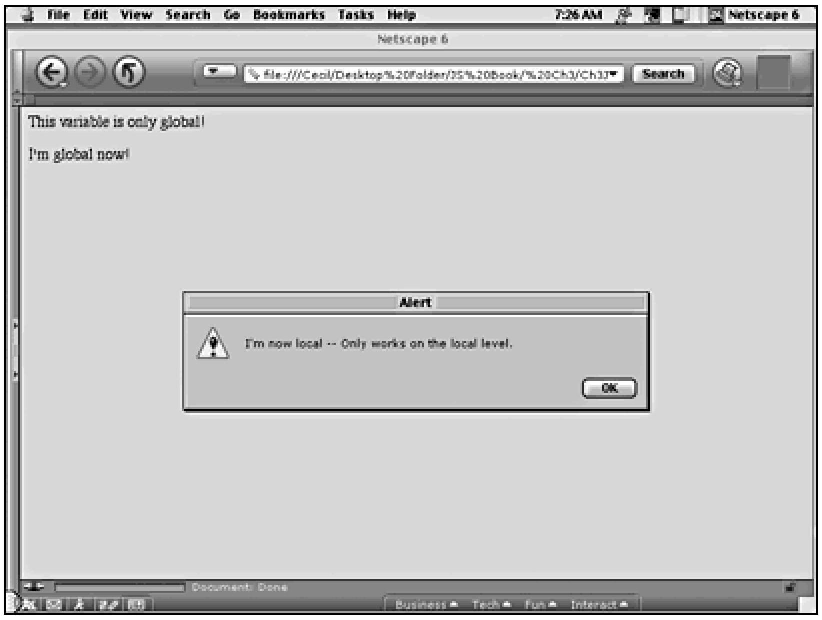
The following script uses four variables to demonstrate these differences. Two global variables are defined, and then two local variables are defined within a function. One of the global and local variables share a common identifier, `localGlobal`. When fired from the function, the local variable's value is displayed; when displayed from the global script, the global variable's value is displayed.

GlobalLocal.html

```
<html>
<head>
<script language="JavaScript">
var onlyGlobal="This variable is only global!";
var localGlobal ="I\'m global now!";
      function showMe( ) {
           var localGlobal="I\'m now local";
           var onlyLocal="Only works on the local level."
           alert(localGlobal + " -- " + onlyLocal);
           }
showMe( );
document.write(onlyGlobal + "<p>"+ localGlobal);
alert(onlyGlobal);
</script>
<body bgcolor=#CadDad>
</body>
</html>
```

Figure 3.4 shows what your page will look like the second time you open it. Initially, you will see only a blank page with the alert box, but after the second alert, you can see both the values from the local and global variables on the screen simultaneously.

Figure 3.4. Global and local variables of the same name can return different values.



```
This variable is only global!

I'm global now!
```

```
I'm now local -- Only works on the local level.
```

In Chapter 6, where functions are discussed in detail, you will learn how to maximize the use of global and local variables in functions. In the meantime, just remember to keep everything between the two types of variables straight by using the `var` keyword in all your variable declarations.

Primitive and Compound Data

Data types are divided into two basic categories, *primitive* and *compound.* Boolean values, numbers, strings, and the `null` and `undefined` values all constitute primitive data types. As you have seen, different data types are handled differently.

Compound data types are made up of more than one component. Two primitive data types, such as 10 multiplied by 7, can make up compound data. Compound data can have components made up of other compound data, made up of other compound data, *ad infinitum.* Compound data can mix and match different data types as well.

In addition to a multiple of primitives, compound data are made up of arrays and objects. By definition, arrays and objects are made up of more than one object. Arrays are a collection of elements, and objects are a collection of properties. JavaScript objects are discussed in detail in Chapter 7, and arrays are examined at the end of this chapter.

In storing the two different data types, JavaScript uses two very different strategies. Primitives are stored in a fixed chunk of memory, depending on the type of primitive data. Because a Boolean literal is either `true` or `false` (`1` or `0`), a Boolean can be stored in a single bit, while a number can take up several bytes. What is important is that primitives have a finite and known amount of space in memory and can be stored with the variables.

Compounds, on the other hand, can get quite complex, as is the case with objects, regular expressions, and functions. Rather than having a space in memory for compound variable values, JavaScript has pointers or references to the values. In other words, the variables themselves are made up of directions to locate their value rather than the actual value.

Calling strings "primitives" can be a bit misleading because they are obviously anything but fixed in size. They are considered *immutable,* meaning that the string value cannot be changed. So, if the value of a string cannot be changed, how can a variable with the value `Tom` be changed to a value `Jerry` ? Obviously, a pointer has to point to the changed value, but strings act like primitives; therefore, I treat them as primitives.

Arrays

Because objects are collections of properties with each property having its own name and value, arrays are actually JavaScript objects. Each property in an array is an *element,* and each element can be assigned a value. One way to think of an array is as a collection of numbered variables.

An array in JavaScript has the following general formats for assigning values to elements:

```
sampleArr[0]=1;
sampleArr[1]="ice cream";
sampleArr[2]=55 * (7 + alpha);
sampleArr[3]= shootTheMoon(73);
sampleArr[4]= otherArr[7];
```

or

```
sampleArr=new Array(1,"ice cream", (55 * (7+ alpha)),
shootTheMoon(73),
otherArr[17]);
```

or, in JavaScript 1.2 or later:

```
sampleArr=[1,"ice cream", (55 * (7+ alpha)), shootTheMoon(73),
otherArr[17]];
```

All three arrays are identical, using different methods for data assignment. The second two methods show the array to be more of an object, while the first method shows the variable-like characteristics of arrays.

In JavaScript, array elements begin with `0` and can be numbered sequentially or nonsequentially. In the second two previous examples, the first element is `0`, and the other data separated by commas are numbered sequentially. However, you could have the following data assignment in an array:

```
alphaArr[0]= "uno";
alphaArr[7]= "dos";
alphaArr[345]= "tres";
```

Usually, arrays are numbered sequentially so that data can be added and extracted using loops. However, an array element can be called forth in any order and used just like a variable.

The data and data types that can be assigned to an array element are identical to the data and data types that you can assign to variables. You will also find that you can assign objects the same kinds of data. (Remember that an array is an object because it is composed for more than a single property.)

Setting Up an Array

Arrays are created using a constructor, just like other objects. The `Array( )` constructor uses the following format:

```
var parts = new Array( );
```

Now, `parts` is an array object, and you can add data as in the following format:

```
parts[0]= "bolts";
parts[1] = "nuts";
```

As with declaring a variable, you need not enter data upon declaring the array, but you can. For example, you may declare and define a dense array using the `Array( )` constructor:

```
var parts = new Array("bolts", "nuts", "washers", "screws");
```

You can also create an array using a *dimension* argument. If you know ahead of time how many elements are in your array, you can declare it and set a dimension for it at the same time. (In some languages, especially older ones, you are required to include a dimension for an array to reserve memory for your array.) For example, if you are creating an array of the U.S. Senate and you know that you have only 100 senators, you could declare and dimension your array as follows:

```
var senators= new Array(100);
```

Once declared and dimensioned, you can add 100 elements, but the *first* senator would be this:

```
senators[0]
```

The last would be this:

```
senators[99]
```

You still get 100 elements, but you must begin with 0 instead of 1.

The declaration using a dimension argument is the exception to the rule that the first data value in a dense array (one in which the data are contained in parentheses) is considered element 0. Element 0 in the array will be the first data assignment after the declaration. For example, the following declaration first dimensions the array at 5 and then enters the first array element (0) as 7 in the next line:

```
var catWeights= new Array(5);
catWeights(7,15,34,52,60);
```

The numeric data value `5` is not a value of any of the array elements. It is the length of the array, with elements ranging from 0–4.

NOTE

Throughout the book, I use the tag `<script language="JavaScript">` *without specifying the version number of JavaScript. You do not need to specify the version number to get the latest version of JavaScript supported by your browser. When using a dimension argument with Netscape Navigator 4 (NN4) and later, you will run into a bug if you specify* `<script language="JavaScript1.2">` *and then attempt to dimension an array. NN4+ treats the declaration with the dimension value as the* first value in the array *instead of the length of the array. For example, try the following script:*

```
<html>
<head>
<script language="JavaScript1.2">
var test= new Array(23)
document.write(test.length);
</script>
</head>
<body>
</body>
</html>
```

When you run the script using NN4 or later, a `1` *appears on the screen. If you remove the* `1.2` *attribute from the* `<script>` *tag, the returned value is* `23`*, the correct length. Because JavaScript 1.5 runs just dandy in Netscape Navigator 6 without specifying the version number in the* `<script>` *tag, and the latest version for Internet Explorer also runs well without specifying the version number, I prefer to leave out all version numbers as a habit. In this way, I can avoid bugs and span more JavaScript and browser versions.*

A final way to declare an array using JavaScript 1.2 or later is called an *array literal.* The declaration uses brackets instead of parentheses and does not require the `Array( )` constructor. The array object name is declared simply by assigning values to it within brackets, as the following script shows:

```
<html>
<head>
<script language="JavaScript">
var Lit= ["yes","no","maybe"];
document.write(Lit[2]);
</script>
</head>
<body bgcolor=#face00>
```

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

The literal array `Lit` looks almost like a variable definition, were it not for the bracketed list of values. Using an array literal saves a couple of steps because no constructor is used, but older browsers (those before support of JavaScript 1.2) will not understand it as an array.

Array Properties and Methods

As an object, arrays have a single property, `length`, and several methods. However, *Netscape Navigator 4* introduced five methods, `Array.pop()`, `Array.push( )`, `Array.shift( )`, `Array.unshift( )`, and `Array.splice( )`, that are not supported either by Internet Explorer or the EMCA-262 standard. To avoid problems but to be inclusive, I have placed the NN4+ array methods in a section at the end of the chapter to alert you to the fact that users of IE will not parse them correctly. Like all JavaScript enhancements not supported by the EMCA standards, I do not recommend using these methods for creating web sites that expect to have viewers using both major browsers.

Array Length

The single array property `length` returns the number of elements in an array. When using a loop, the test condition for the loop can be the length of the array so that you need not use an invariant value for the test. The format is as follows:

```
Array.length
```

The property is easily passed to a variable, as the following sample shows:

```
var dogs = new Array("Beagle","Terrier","Collie","Mutt");
var dogTail= dogs.length;
```

The variable `dogTail` would have a length of `4` because four elements make up the array. The `length` property *does not* refer to the number of characters that make up the element in the array, but it refers to the number of elements themselves. Thus, the following array has a length of `2`, even though more characters are used than in the first example with a length of `4`:

```
var dogs = new Array("Greater Swiss Mountain Dog","Irish Wolfhound");
```

Concatenating the Elements of an Array: join(), toString(), *and* concat()

The `Array.join( )` method takes all of the values in all of the elements in the array and creates one big string. For example, try out the following script:

joinArray.html

```
<html>
<head>
```

```
<script language="JavaScript">
var trees= new Array("Elm","Pine","Oak");
var bigBush=trees.join( );
document.write(bigBush);
</script>
</head>
<body bgcolor=#ace007>
</body>
</html>
```

The results are the contents of the array minus the quotation marks, showing you `Elm,Pine,Oak`.

The `Array.join( )` method accepts an argument that acts as a separator. Whatever you place in the `join( )` parentheses within quotation marks replaces the commas. For example, change this line:

```
var bigBush=trees.join( );
```

to

```
var bigBush=trees.join(" and ");
```

Then, launch the script again for a different result. The second results are `Elm and Pine and Oak`.

An older method from JavaScript 1.1 that is similar to the `Array.join( )` method is `Array.toString( )`. The `toString( )` method generates the same results as `join( )`, but you cannot specify the connecting characters between elements as you can with `join()`.

A third method used for concatenating string elements in arrays is `Array.concat()`. Not only does the `concat()` method join all existing elements, but it also adds the elements in a `concat()` argument. For example, by changing the joinArray.html script slightly, you can see how it works.

concatArray.html

```
<html>
<head>
<script language="JavaScript">
var trees= new Array("Elm","Pine","Oak");
var biggerBush=trees.concat("Maple", "Sycamore");
var bigBush=trees.join();
document.write(biggerBush);
alert(bigBush);
</script>
</head>
<body bgcolor=#ace007>
</body>
</html>
```

A very important part of the concatArray.html script is that you can see that the `Array.concat()` method *does not change the contents of the array.* The array named trees still has only three elements. That is evidenced by the fact that the alert message shows only three elements, even though the variable `bigBush` was defined *after* the `biggerBush` variable was defined and added the Maple and Sycamore data to the mix. To add elements to an array, you assign new values to named elements. For example, to include the Maple and Sycamore data to the array, you could write this:

```
trees[3]="Maple"; // The fourth element is 3 since the first is 0
trees[4]="Sycamore";
```

Changing the Order of an Array: sort() *and* reverse()

Two methods are available to change the order of array elements. The first sorts the array string elements alphabetically, and the second reverses their order.

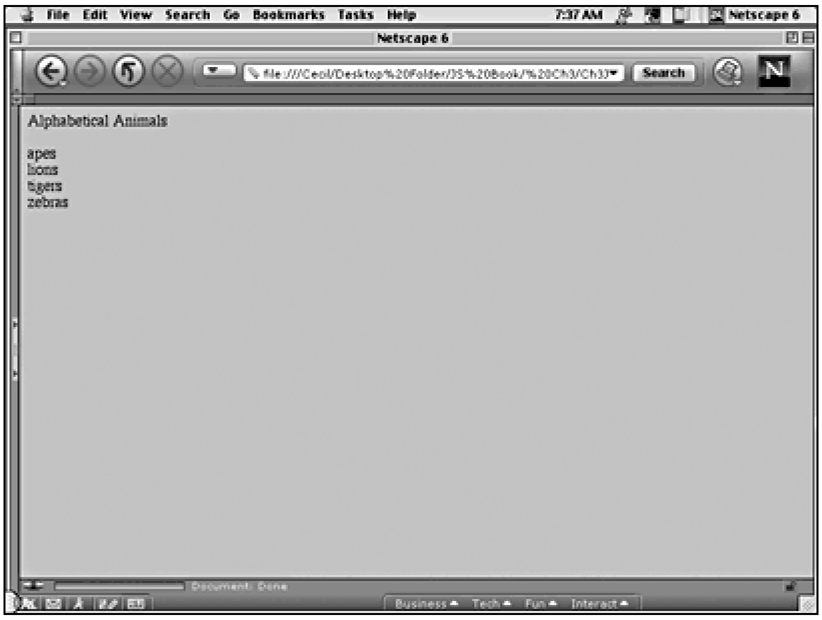
The `Array.sort()` method is very simple, especially when using strings. After entering all of the strings in the array, you just enter the name of the array and method, and the array is ordered alphabetically. The following example shows both how the `sort()` method works and how array elements are extracted with a loop statement.

sortArray.html

```
<html>
<head>
<script language="JavaScript">
var zoo= new Array("zebras","lions", "apes","tigers");
zoo.sort( );
var newZoo="";
for(var counter=0; counter<zoo.length; counter++) {
newZoo += (zoo[counter] + "<br>") ;
}
document.write("<p>Alphabetical Animals<p>" + newZoo);
</script>
</head>
<body bgcolor="lightsteelblue">
</body>
</html>
```

As you can see in Figure 3.5, all of the values are arranged in alphabetical order.

Figure 3.5. By placing string data into an array, you can easily sort it using the `Array.sort( )` method.


Alphabetical Animals

apes
lions
tigers
zebras

```
apes

lions

tigers

zebras
```

For ordering lists of any kind, you will find the `Array.sort( )` method a handy tool.

The `Array.reverse( )` method simply reverses the order of the data in the array. The first element becomes the last element, and everything else in the array is reversed as well. For example, the following:

```
var majorCities=new Array("Tokyo", "Los Angeles", "Paris", "Beijing",
"Bloomfield")
marjorCities.reverse( );
```

would return this:

```
Bloomfield, Beijing, Paris, Los Angeles, Tokyo
```

By using `Array.sort( )` and `Array.reverse( )` in concert, you can change ascending and descending orders of a sorted list.

Extracting Subarrays: slice()

To specify a subarray, use `Array.slice( )`. The general form of using `slice( )` is shown here:

```
ArrayName.slice(begin,end)
```

or

```
ArrayName.slice(begin to end)
```

For example, if you have these statements:

```
computer=["Dell", "Gateway","Apple","IBM","HP"];
computer.slice(2,4);
```

your return would be

```
Apple,IBM
```

Using a single argument takes the element from the argument to the end of the array as defined by the argument. For example:

```
computer.slice(2);
```

would return

```
Apple,IBM,HP,
```

Negative numbers constitute a final type of argument used in the `Array.slice( )` method. The negative value begins with the *last* element in the array as −1 and then counts backward toward the first element. For instance, the statement from the example in this section:

```
computer.slice(-1)
```

returns this:

```
HP
```

Unlike the forward-counting slices that begin with 0, the last element in an array is identified as −1 using the `slice( )` method with an array.

Navigator 4 Core Array Methods: pop(), push(), shift(), unshift(), splice()

This last set of methods adds a good deal of utility to working with arrays, and you can perform stack-like operations with the array. If you've ever written programs in Forth or written code for Adobe PostScript, you've worked with the stack, and `pop( )` and `push( )` are familiar. Each of the five is described briefly with a short explanation and is based on the following single example:

```
var stackWork= new Array("Lenny","Harold","Mary","Jean", "Sal");
```

`Array.pop( )` removes the *last* element of an array and returns it.

`stackWork.pop( );` returns `Sal` and removes it from the array. If a second identical statement were made on the next line, it would return `Jean`.

`Array.push( )` adds a value to the end of the array (top of the stack) and leaves it there. By adding the following line, the string `Delia` would become the value to a new last element added to the array by the `push( )` method:

```
stackWork.push("Delia");
```

In a `pop( )` operation, it would be the first one off (LIFO—last in, first off).

`Array.shift( )` removes the *first* element in an array and returns it. For example, this would return `Lenny`, remove it from the array, and shift the remaining elements to the left:

```
stackWork.shift( );
```

Element 0 would become `Harold`.

`Array.unshift( )` is similar to the `push( )` method, except that the new element is put at the front of the array (bottom of the stack). If you entered the following, the first element (element 0) in the array would have a value of `Willie`, the value `Lenny` would be shifted to the right into element 1, and so on for the entire array:

```
stackWork.unshift("Willie");
```

Finally, `Array.splice( )` is a method used to insert, delete, and substitute values in array elements. The method has three arguments, `start`, `delete`, and `data`. The starting position specifies where the new value (data) is to be inserted and where deletions begin. If no deletions are specified, the `splice( )` method has the effect of inserting an element and value into an array. For example, the following shows how to insert the value Fred into the second element, leaving the first as it is and shifting the rest to the right:

```
stackWork.splice(1,0,"Fred");
```

The `splice( )` method allows you to insert elements, delete one or more elements, or change the value of an array element. You might find that these functions do not work in some of the older browsers, but *IE5*+ and *NN4*+ work well with them.

Summary

This chapter examined both the data types and the containers for data in JavaScript. An understanding of data types and variables in JavaScript is essential to working effectively with JavaScript because changes in the properties and objects that make up JavaScript and HTML are dependent on controlling the values in variables. Because the *values are the data,* all of the work with objects depends on understanding how to use the data types and their containers, variables.

In addition, array objects, along with the property and methods associated with arrays, were introduced. Each element of an array has variable-like characteristics, but an array is a very different breed of animal. With an object, the web designer can use the built-in property and methods to order and change the array elements. In further chapters, as more structures in JavaScript are revealed, you will see far more applications for and the value of arrays.

Chapter 4. Using Operators and Expressions

CONTENTS>>

The previous Chapters Made Extensive Use of both operators and expressions. This chapter examines the operators and the expressions that they create. In programming, the name “expression” is a bit misleading because both literals and a combination of literals and variable names can be expressions. Thus, this excerpt:

```
total
```

is an expression in a line of JavaScript code, as is this:

```
total + tax + 4.99
```

So, while an expression is usually envisioned as a combination of literals or variables (complex expressions), expressions can be single variables or literals.

General and Bitwise Operators

To begin an examination of operators and expressions, two very different types of operators are discussed separately. I will refer to nonbitwise operators as “general operators” or simply “operators,” while bitwise operators are prefaced by “bitwise.” In this next section, Table 4.1 shows an overview of what I call general operators; near the end of the chapter, you will find a separate subsection and Table 4.2 on bitwise operators.

Bitwise operators shift the binary numbers (0s and 1s) in the registers in your computer. When binary numbers are shifted, special types of mathematical operations occur that you normally associate with addition, subtraction, multiplication, and division. To learn more about how this works, consult a good book on binary math. Generally, though, designers do not need to use binary operators or binary math, so if you skip over the section on binary operators, you probably will not miss anything crucial to creating dynamic pages with JavaScript.

General Operators in JavaScript

The everyday garden-variety operators seen in a typical JavaScript program are very similar or identical to the operators found in programs such as C++ or Java. If you have a Basic or Visual Basic programming background, you will find many similarities as well between JavaScript operators and what they use in different types of Basic. However, while a quick glance at the operators shows where the operators are similar and where they differ from your previous programming experience, be sure to take that glance. For readers who are taking up JavaScript as their first language, spend some time going over the examples and descriptions of what the operators do (see Table 4.1).

Table 4.1. JavaScript Operators

Symbol	Operation
+	Add or concatenate
-	Subtract, or negative
*	Multiply
/	Divide
%	Modulus (remainder)
++	Increment
-	Decrement
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
&&	Logical AND
||	Logical OR
!	Logical NOT
= =	Test for equality
= = =	Test for strict equality
! =	Test for inequality
! = =	Test for strict inequality
=	Assignment
+ =	Add and assign
- =	Subtract and assign
* =	Multiply and assign
% =	Modulus and assign
/=	Divide and assign
( )	Function arguments
.	Structure member (called a dot) in structure or property
,	Multiple evaluation
[ ]	Array elements (access, index)
,	Multiple evaluation

`? :`	Ternary operator
`new`	Constructor
`delete`	Remove a variable
`typeof`	Data type
`void`	Return undefined value

Operators

Operators can be placed into three categories—*binary, unary,* and *ternary.* Binary operators, most commonly associated with the concept of *operator,* take *two* (binary) expressions and combine them into a third complex or compound expression. However, a single expression can have several binary operators. For example, the following variable declaration uses multiple binary operators to define the variable:

```
var calcAdd = (total / n ) + 73
```

The divide (`/`) operator and the plus (`+`) operator are binary operators. The first combination occurs when the variable `total` is divided by the variable `n`. The two variables become a single value. That single value resulting from `total` divided by `n` is then added to the literal numeric value of 73, creating yet another value. The equals sign (`=`) places the combined value of the operands into the variable `calcAdd`.

Unary operators work on a single variable or literal. All negative numbers are assigned using a unary operator. For example, the following little script uses a unary operator to create a variable with a negative value:

```
<html>
<head>
<script language="JavaScript">
var posNum=85;
var negNum= -posNum;
document.write(negNum);
</script>
</head>
<body bgcolor="honeydew">
</body>
</html>
```

The return of the script is `-85` because the minus (`-`) unary operator defined the variable `negNum` as the negation of the variable `posNum`. Other common unary operators include increment or decrement operators (++ and - -) seen in counter variables.

Finally, ternary operators combine three expressions into one. Most commonly used to create a shorthand expression for conditional statements, the only ternary operator in JavaScript is `? :`. For example, this conditional statement:

```
if(alpha == beta) {
      gamma=56;
      } else {
```

```
      gamma=57;
      }
```

can be written with a ternary operator as follows:

```
alpha == beta ? gamma=56 : gamma=57;
```

The following little script shows how both methods arrive at the same conclusion:

```
<html>
<head>
<script language="JavaScript">
var alpha=20, beta=30, gamma=0, lambda=0;
if (alpha==beta) {
var gamma=56;
} else {
gamma=57;
}
//Same set of conditions using ternary operator
alpha==beta ? lambda=56 : lambda=57;
document.write("Conditional results:" + gamma + "<p>" + "Ternary
conditional:" + lambda);
</script>
</head>
<body bgcolor="oldlace">
</body>
</html>
```

The three elements that the `?:` operator brought together in the example are `(alpha==beta)`, `(lambda=56)`, and `(lambda=57)`. Note also how the comma (`,`) operator is used in the script to separate the definitions of the variables `alpha`, `beta`, `gamma`, and `lambda` at the beginning of the script.

Assignment Operators

The key assignment operator is the equals sign (`=`). The left operand is a variable, an array element, or an object property, and the right operand is either a literal or another variable, array element, or object property. As seen in Chapter 3, “Dealing with Data and Variables,” assigning a variable a value can be accomplished with any number of different combinations of variables, array elements, object properties, and literals.

The following provides an idea of the range of assignments:

```
alpha= 77;
alpha= (fishSize.length / 2);
alpha= (beta > gamma);
```

Compound Operators

Operators that include assignment along with an operation are *compound operators.* These operators work as a shorthand for an assignment plus another operation. For example:

```
var bankAccount += interest;
```

is equivalent to writing

```
var bankAccount = bankAccount + interest;
```

Besides addition, compound operators in JavaScript include subtract assign (`-=`), multiply assign (`*=`), divide assign (`/=`), and modulo assign (`%=`). For example, the following script uses the modulo compound assignment operator:

```
<html>
<head>
<script language="JavaScript">
var bolts=150, lot= 60;
bolts %= lot;
document.write("Odd lot=" + bolts);
</script>
</head>
<body bgcolor="lightslategray">
</body>
</html>
```

The example shows how two operations can be combined into a single one. The variable `bolts` is divided by the value of the variable `lot`, and the remainder (modulo) is assigned to the variable bolts. It would be the same as writing this:

```
var bolts = bolts % lot;
```

However, instead of taking two operations, one does the trick of assignment and operation.

Comparison Operators

Probably the area of most mistakes in JavaScript with operators is confusing (or just forgetting) the difference between assignment operators and comparison operators. Assignment operators equate a value with a variable, array element, or object property. Comparison operators generate a Boolean value. For example, the following script returns a `false` Boolean value:

```
<html>
<script language="JavaScript">
var wrong= (6==7)
document.write(wrong);
</script>
<body bgcolor="lightslategray">
</body>
</html>
```

The comparison operator is the double equals sign (`= =`), and the assignment operator is the equals (`=`) sign. The most common problem is in a standard conditional statement where the developer types this:

```
if (alpha = beta) { .... WRONG
```

when he meant to type this:

```
if (alpha==beta) { .... RIGHT
```

In the debugging process, one of the first things to look for is the placement of an assignment operator where an equality operator should be.

The other comparison operators include not equal to (`!=`) less than (`<`), greater than (`>`), less than or equal to (`<=`), and greater than or equal to (`>=`). Like the equality operator, these other comparison operators have two roles. One role is part of a conditional statement, and the other is to serve as Boolean literals in definitions as the previous example showed. The following script shows how variables can be defined so that they can contain Boolean literals and then used as part of a conditional statement *without* the use of comparison operators:

```
<html>
<script language="JavaScript">
var alpha=25;
var beta=35;
var zeta=(alpha <= beta);
if(zeta) {
     var sigma ="This is true."
     } else {
     var sigma = "This is not true.";
     }
document.write(sigma);
</script>
<body bgcolor="lightslategray">
</body>
</html>
```

In the script, the variable `alpha` is compared to be less than or equal to (`<=`) `beta` in the definition of `zeta`. Because `alpha` is less than `beta`, the variable `zeta` contains a Boolean value of `true`. In the conditional statement, *no comparative operators are used* because the variable `beta` is already a Boolean value. Because the value is `true`, it meets the condition to load the variable sigma with the message "This is true."

Strict Equality Operators

JavaScript 1.3 introduced strict equality and inequality operators. These operators test for both equality of value and *equality of type.* In other words, if both values were `23` but one variable is a string and the other is a number, you might think that they would be unequal anyway. Consider the following script:

```
<html>
<script language="JavaScript">
var currentWord="75";
var currentNumber=75;
var outcome=(currentWord==currentNumber);
```

```
document.write(outcome);
</script>
<body bgcolor="lightsalmon">
</body>
</html>
```

You might be surprised to find that the variable outcome is `true` ! The reason for that is that JavaScript tries very hard to resolve numeric and string differences. Remember, if you define a variable as follows, the outcome is a string even though the line mixes numeric and string literals:

```
var mix = "$" + 12.33;
```

The same is true when JavaScript compares two variables where one is a number and one is a string. If the "values" are deemed to be the same even though one is a string and the other is a number, JavaScript helpfully makes them equal, as was seen in the previous script. However, if you had an application where *both* the values *and* the type of data are important to compare, you could not make that comparison with standard comparison operators. To fix that problem, *JavaScript 1.3* introduced strict equality (`===`) and inequality (`! ==`) operators. These operators look at not only the values, but also at the type of variable. In the previous script, change this line:

```
var outcome= (currentWord==currentNumber);
```

to

```
var outcome= (currentWord===currentNumber);
```

Then save the script and run the program again. In the second version of the script, the outcome changes to `false`. While the numbers are the same, the data types are different. To get a `true` outcome, change the line to the following:

```
var outcome= (currentWord! ==currentNumber);
```

Both *Netscape Navigator 4.7* and *Internet Explorer 5* and later recognize strict equality and inequality operators. (Version 4 of Netscape Navigator requires `language=JavaScript1.2` in the `<SCRIPT>` tag, but in later versions of the browser, all you need is `language=JavaScript`.)

Arithmetic Operators

The basic arithmetic operators in JavaScript are fairly self-explanatory, with a few exceptions. To avoid the few aggravations from these exceptions, each operator is discussed with a focus on uses.

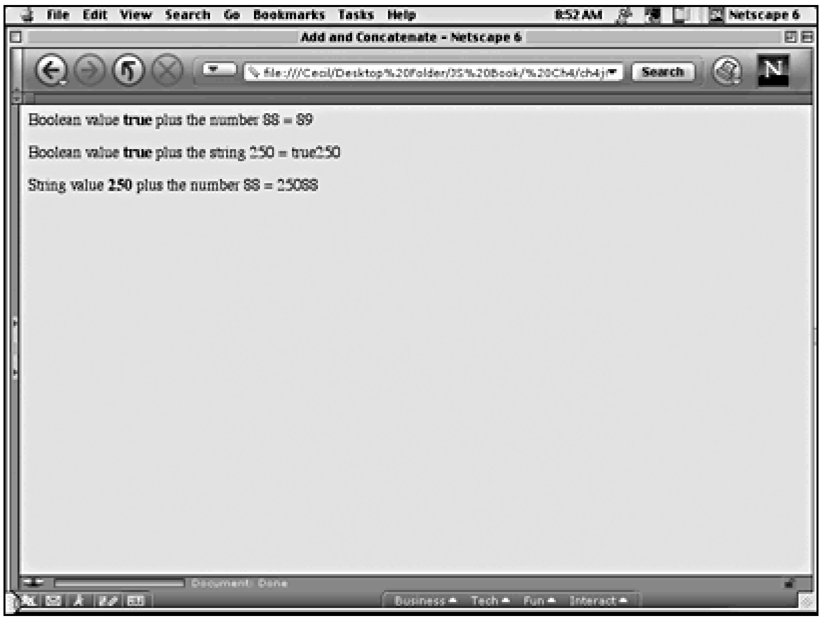
Add and Concatenate (+)

One arithmetic operator that has two different uses is add (`+`). First, the add operator adds values in math operations. Second, it concatenates (joins) strings or strings and other literals. Mathematical addition is fairly straightforward, but concatenation is not. When the add operation joins a string and a number, it concatenates them *and* converts the number into a string. For example, the following script joins strings with nonstring literals:

```
<html>
<head>
<title>Add and Concatenate</title>
<script language="JavaScript">
var Boole= 22 < 90;
var string="250";
var numnum=88;
var BooleNum =Boole + numnum;
var BooleString=Boole + string;
var StringNum=string + numnum;
var part1="Boolean value <b>" + Boole + "</b> plus the number " +
numnum + " = " +
BooleNum;
var part2="<p>Boolean value <b>" + Boole + "</b> plus the string " +
string + " = " +
BooleString;
var part3="<p>String value <b>" + string + "</b> plus the number " +
numnum + " = " +
StringNum;
document.write(part1+part2+part3);
</script>
</head>
<body bgcolor="paleturquoise">
</body>
</html>
```

The output seen in Figure 4.1 tells the story of what happens when different types of literals are mixed. Whenever a number or a Boolean value is added to a string, it is turned into a string and is concatenated with the string. However, when a Boolean value is added to a number, the value of the Boolean (`0` or `1`) is added to the number, and the Boolean value is treated *as a number* instead of `true` or `false`. However, when a Boolean and string are combined, the Boolean value is shown as a string `true` or `false`.

Figure 4.1. Depending on the use of the add (+) operator, different results can be expected.


File Edit View Search Go Bookmarks Tasks Help
8:52 AM
Netscape 6
Add and Concatenate - Netscape 6
Search
Document: Done
Business Tech Fun Interact

Boolean value **true** plus the number 88 = 89

Boolean value **true** plus the string 250 = true250

String value **250** plus the number 88 = 25088

Subtract and Negation (-)

The minus sign (-) has two very different uses. First, in arithmetic operations, the subtract operator subtracts the second operand from the first. Hence, this line places the value 7 in the variable alpha:

```
var alpha = 10-3;
```

Second, used as a unary operator, the minus sign changes a positive value to a negative value. Moreover, if a negative value is subtracted from a positive value, the result is the same as adding two positive values. The following script illustrates using both the unary negation and subtraction with the minus (-) sign as an operator. For example, try out the following script and see if you can determine ahead of time what the outcome will be:

```
<html>
<head>
<title>Minus sign and negative values</title>
<script language="JavaScript">
var posVal=44;
var negVal= -posVal;
var diffVal = (posVal-negVal);
document.write(diffVal);
</script>
</head>
<body bgcolor="papayawhip">
</body>
</html>
```

If you guessed that the output on the screen would be 88, you are right. The positive value in the variable posVal is 44. The variable negVal is created by the unary negation of posVal. When negVal is subtracted from posVal, the effect is to add the two values. (Just as in grammar, a double-negative creates a positive.)

Multiply (*)

The multiplication operator is simple—it multiplies two numbers. However, if you attempt to multiply two strings containing numeric characters, JavaScript attempts to change the strings into numbers and complete the multiplication. For example, try the following script:

```
<html>
<head>
<title>Multiply Numbers in Strings</title>
<script language="JavaScript">
var stringNum="5";
var stringNum2="20";
var mulEm= stringNum * stringNum2;
document.write(mulEm);
</script>
</head>
<body bgcolor="peru">
</body>
</html>
```

The output to the screen will be `100`. So, the multiply operator (`*`) can actually convert certain strings into numbers as well as multiply numbers.

Divide (/)

Like the multiply operator, the divide operator (`/`) works with numbers. In operation, the left operand is divided by the right operand. Also, like the multiply operator, the divide operator attempts to convert a string into a number. The area where division differs most from the other operations is in a divide by zero error. Two different types of returns result. A divide by zero of a number other than 0 results in `Infinity`, while zero divided by zero returns `NaN`. The following script demonstrates what returns from both types of divide by zero errors. Also, the script shows how to use the built-in functions `isFinite()` and `isNaN()` to test for `Infinity` and `NaN` values. In the case of `Infinity`, the `isFinite()` function must be negated using the `!` operator.

For a designer, the importance of knowing when a divide by zero instance occurs is so that you can keep it from crashing your program. A home-decorating site, for example, has a module that calculates the amount of paint required to paint rooms. A gallon of paint covers 350 square feet. So, somewhere in the calculator, the designer must have a formula that divides 350 by the square feet of the room being painted. Suppose that the viewer forgot to enter a value for the square feet of the room and the script attempted to divide 350 by 0. Rather than sending the viewer into confusion by telling her that she had to purchase an infinite number of gallons of paint, you can trap to divide by zero errors and send any message that you want, as the following script illustrates:

```
<html>
<head>
<title>Divide by Zero Errors</title>
<script language="JavaScript">
var leftOperand=77;
var rightOperand= 20 > 30;
var divEm= leftOperand / rightOperand;
var nada= 0/0;
document.write(divEm + "<p>" + nada);
if(!isFinite(divEm)) {
alert("Whoa Dude that\'s a big number!")
}
if(isNaN(nada)) {
alert("You are dividing nothing by nothing.")
}
</script>
</head>
<body bgcolor="springgreen">
</body>
</html>
```

Modulo (%)

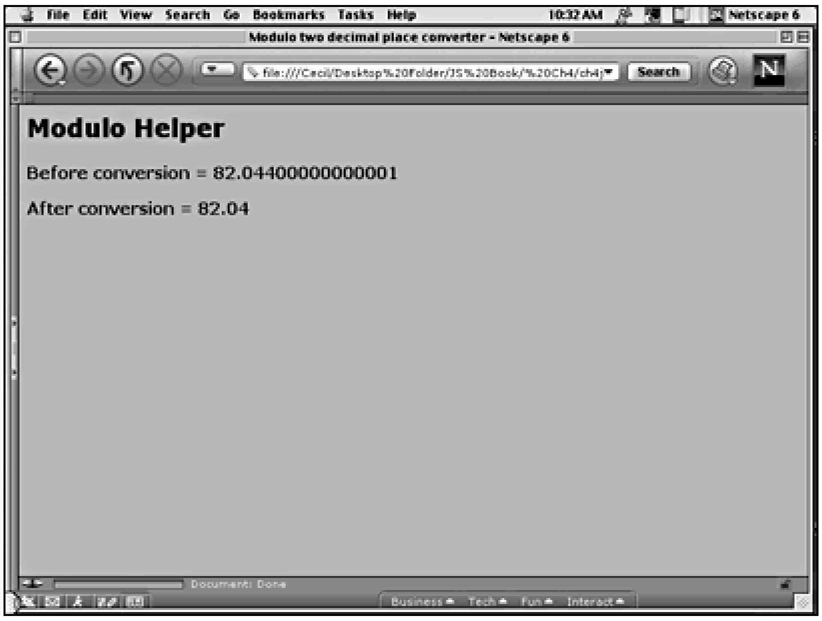
The modulo (`%`) operator returns the remainder in a division operation. The left operand is divided by the right operand, and *only* the remainder is returned. While the modulo operator does not come to mind in most applications, it can prove to be an extremely useful operator. For example, the following script uses the operator to convert long decimal places into two-place decimals.

modulo2dec.html

```
<html>
<head>
<title>Modulo two decimal place converter</title>
<style>
body {
background-color: plum;
font-family: verdana;
font-weight: bold;
}
</style>
<script language="JavaScript">
var dec=.06;
var part=77.4;
part += (dec * part);
var wholeInt=Math.floor(part);
//The Math.floor() function rounds the variable 'part' down to the
nearest whole integer.
//Prior to obtaining the modulo (remainder) 'part' times 100 is
rounded down to the
nearest whole to obtain an integer remainder.
var fraction=Math.round(part *100)%100;
if (fraction<10) {
fraction = "0" + fraction;
}
var fullVal=wholeInt +"." + fraction;
var headTitle="<h2>Modulo Helper</h2>"
var before="Before conversion = " + part + "<p>";
var after="After conversion = " + fullVal;
document.write(headTitle + before + after);
</script>
</head>
<body>
<center>
</body>
</html>
```

Figure 4.2 shows what the original number looks like before and after the conversion script helped along using modulo.

Figure 4.2. You can use the modulo operator to help write a script to round numbers to two decimal places.


File Edit View Search Go Bookmarks Tasks Help
10:32 AM
Netscape 6
Modulo two decimal place converter - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/%20Ch4/ch4j
Search
Document: Done
Business Tech Fun Interact

Modulo Helper

Before conversion = 82.04400000000001

After conversion = 82.04

Increment (++) and Decrement (- -) Operators

These operators either add 1 or subtract 1 from an operand. In examples where loops have been used, the `counter` variable typically increments or decrements using these two operators. This general form in a loop statement is the most common usage of the increment or decrement operator:

```
for (counter=0; counter<20; counter++) {....
```

The operand connected with these two operators can be preaffected or postaffected. If the operator is in front of the operand, the value is added or subtracted *before* the next operation. If the operator is at the end of the operand, the addition or subtraction comes *after* the operation. For example, the following script can be used to show how each affects the operand:

```
<html>
<head>
<title>Increment/Decrement operator</title>
<script language="JavaScript">
var combine="";
var bounce=0;
for (var counter=0;counter <=5; counter++) {
      var Hep=bounce++;
      combine += "Hep value =" + Hep + "<br>";
      }
document.write(combine);
</script>
</head>
<body bgcolor="palevioletred">
```

```
<center>
</body>
</html>
```

When you run the script, the outcome on the screen is as follows:

```
Hep value =0
Hep value =1
Hep value =2
Hep value =3
Hep value =4
Hep value =5
```

The first time through the loop, the variable `bounce`, originally declared with a value of zero (0), remained zero because the increment in its value is *after* the definition of the variable `Hep`. Now change the position of the increment operator to the front of the variable, changing the line to this:

```
var Hep= ++bounce;
```

Now the output shows this:

```
Hep value =1
Hep value =2
Hep value =3
Hep value =4
Hep value =5
Hep value =6
```

As can be seen, the position of the increment operator made a fundamental change in the output. With the increment operator in front of the operand, the `Hep` variable was incremented in the first iteration, but it wasn't until the second iteration that the `Hep` variable changed when the operator was at the end of the operand. A small change in the code led to a big change in the output. With increment and decrement operators, you must be especially vigilant not to crash a program because the position of the operator is positioned incorrectly.

Operators in the Context of Using String Variables and Literals

As you saw when using the plus (`+`) operator, numbers can be added or strings and numbers can be concatenated into a single string. So, the idea of a "string operator" is very much a context-dependent concept.

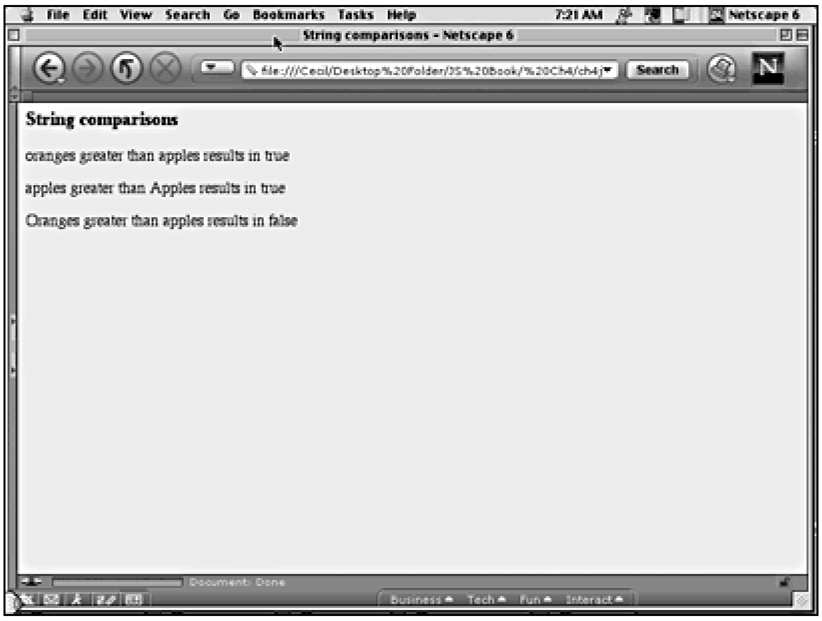
Besides using the plus (`+`) operator, you can use the comparison operators (`>`, `>=`, `<`, `<=`, `==`, `!=`) with strings. In using comparison operators, the operator compares the string operands in an alphabetical order based on Unicode character encoding. The higher in the alphabet the character is, the *greater* the character is in comparison with another character. However, uppercase letters are *less than* lowercase letters. Therefore, `Xray` is less than `emergency`, as far as JavaScript is concerned. The following script shows some relationships between order and uppercase and lowercase strings.

stringOps.html

```
<html>
<head>
<title>String comparisons</title>
<script language="JavaScript">
var alpha="apples";
var beta="oranges";
var gamma="Apples";
var delta="Oranges";
var lclc=beta > alpha;
var lcuc=alpha > gamma;
var uclc=gamma > alpha;
var banner="<h3>String comparisons</h3>"
var first=beta + " greater than "+ alpha + " results in " + lclc +
"<p>";
var second=alpha + " greater than "+ gamma + " results in " + lcuc +
"<p>";
var third=delta + " greater than "+ alpha + " results in " + uclc;
document.write(banner+first+second+third);
</script>
</head>
<body bgcolor="mistyrose">
</body>
</html>
```

Figure 4.3 shows the outcomes that you can expect when working with upperand lowercase comparisons in JavaScript.

Figure 4.3. Strings beginning with lowercase letters are considered greater than those beginning with uppercase letters.


File Edit View Search Go Bookmarks Tasks Help
String comparisons - Netscape 6
Search
Document: Done

```
String comparisons

oranges greater than apples results in true

apples greater than Apples results in true

Oranges greater than apples results in false
```

When comparing strings with numbers, a different outcome occurs than when using the plus (+) operator. Instead of turning numbers into strings, JavaScript attempts to turn strings into numbers when making comparisons that involve numeric characters in strings. For example, if you write the following, the variable `alpha` would be `true`:

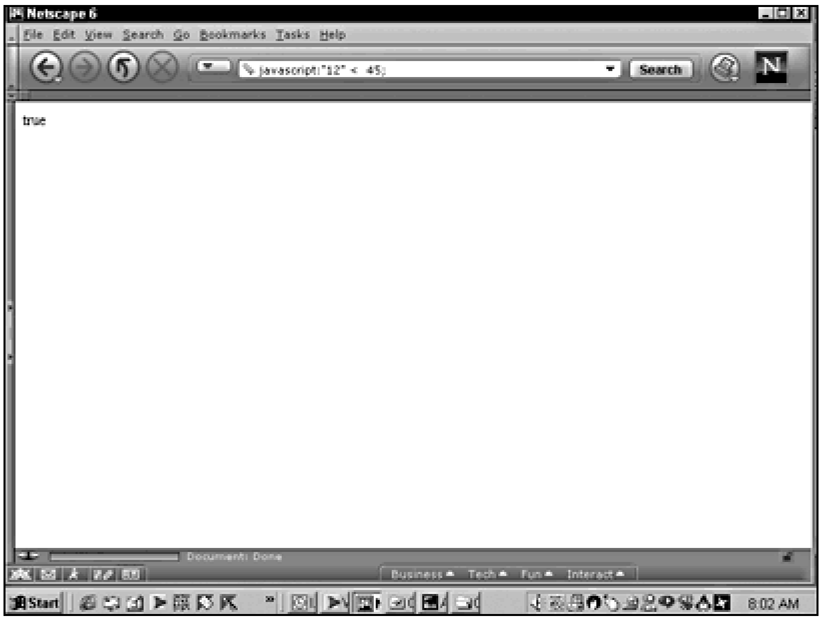
```
var alpha="10" > 3;
```

However, a string with a number followed by character letters does not ignore the letters and make a valid numeric comparison with a numeric operand.

TIP

Whenever you're not sure about what is greater than or less than some combination of numbers, numbers and strings, or strings and strings, use your browser address window as a test bench. Just enter the word `javascript:`*, followed by the operands and operators.* *Figure 4.4* *shows a simple example.*

Figure 4.4. You can use the URL window of your browser to test the outcome of different combinations of numbers or strings using comparison operators.


Netscape 6
File Edit View Search Go Bookmarks Tasks Help
Search
Document: Done
Business Tech Fun Interact
Start
8:02 AM

```
javascript:"12" < 45;
```

```
true
```

Boolean Operators

The comparison operators result in Boolean outcomes, but three logical operators in JavaScript can be considered Boolean as well. The operators combine different conditions or the negation of a condition.

Logical AND (&&)

A common requirement in a script is for two different conditions to exist for an outcome to be `true` or `false`. JavaScript provides the logical AND (`&&`) operator to determine whether two or more conditions are met. For example, an array search might seek to find all instances of customers who are interested in purchasing a new printer *and* who live in the state of Iowa so that they can be contacted for a printer trade show in Des Moines. Only if *both* conditions are true will the outcome be true and added to a contact list. For example, the following script segment searches for two conditions in an array:

```
for (var seek=0;customers.length;seek++)
if((interest[seek]=="printer") && (state[seek] =="Iowa")) {....
```

Note that a double set of parentheses have to enclose the script within the `if` statement. You may also use the logical AND in defining variables. For example, in the following script, the first variable evaluates to `true` and the second evaluates to `false`:

```
<html>
<head>
<title>String comparisons</title>
<script language="JavaScript">
var alpha = (15 < 20) && ("pen" > "Sword");
var beta = ("big" > "tall") && (20 < 30);
document.write(alpha + "<br>" + beta)
</script>
</head>
<body bgcolor="lightcoral">
</body>
</html>
```

Logical OR (||)

The logical OR operator (`||`) uses the double pipes as a symbol. When two or more conditions are stated using the logical OR operator, only *one* of the conditions need be met for the outcome to evaluate as `true`. For example, the variable `alpha` in the following would evaluate to `true`, even though two of the conditions are `false`:

```
var alpha = (56 < 34) || (10 > 2) || ("Fred" > "Alice");
```

You may also use the logical OR (or the logical AND) with variables defined with Boolean values. For example, the following lines show how you might use the logical OR in a script:

```
var alpha = ("beans" > "Potatoes");
var beta = 30 > 40;
var gamma = alpha || beta;
```

Because the variable `alpha` contains a true Boolean value and `beta` contains a false one, the variable `gamma` is `true` because *either* one or the other has to be true, not both.

Logical NOT (!)

JavaScript's logical NOT (`!`) serves to negate an outcome. Sometimes, as used elsewhere in this chapter, a built-in function has the opposite of what you might want to test in your script. The `isFinite( )` function used in an example elsewhere in this chapter was negated to test for `Infinity`. The following script shows some different applications of the logical NOT:

```
<html>
<head>
<title>Logical NOT</title>
<script language="JavaScript">
var alpha = 200/0;
var beta = !isFinite(alpha);
var gamma = !(!alpha);
var delta = !beta;
var b="<br>";
combine =
"alpha="+alpha+b+"beta="+beta+b+"gamma="+gamma+b+"delta="+delta;
document.write(combine);
</script>
</head>
<body bgcolor="mintcream">
</body>
</html>
```

The script generates the following output:

```
alpha=Infinity
beta=true
gamma=true
delta=false
```

Because the `alpha` value generates `Infinity`, `beta` should generate `true` because the function `!isFinite()` tests for `Infinity`. However, the `gamma` variable also generates `true`. The negation of a variable containing a true Boolean literal generates `false`, but so will the negation of any other variable with a non-Boolean value. For example, these lines would return `false`:

```
var alpha=5, beta=!alpha;
document.write(beta);
```

Because `alpha` does not contain a Boolean value, you might assume that `alpha` would be "neutral"—neither true or false. However, in the script where `gamma` returns `true`, a double NOT preceded it— `!(!alpha)`. That is because `!alpha` would generate a Boolean `false`.

Bitwise Operators

If your script calls for bitwise operations, you can use the symbols in Table 4.2 to guide you. Generally, few programmers require bitwise operators, and they are only included here for a complete list of operators available in JavaScript and for programmers who may need them.

NOTE

Bitwise operations involve binary numbers, and you should understand how and when to effectively use binary numbers in a program. However, you can go through life as a very effective programmer, not to mention designer, and never have cause to use bitwise operators. However, if using bitwise operators is crucial *to an envisioned JavaScript program that you have in mind, you will find that JavaScript provides an ample set of bitwise operators.*

Table 4.2. Bitwise Operators

Symbol	Operation
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left-shift
>>	Right-shift
>>>	Zero extension in right-shift

In certain respects, bitwise operators are like any others in the sense that you use them in the same kinds of expressions as any other operators in JavaScript. The key difference is that they work with binary (0s and 1s) instead of decimal values. To see what JavaScript is doing with bitwise operators, consider the first seven values in a binary number system:

```
0000 -0
0001 -1
0010 -2
0011 -3
0100 -4
0101 -5
0110 -6
```

If the digits in one were shifted to the left by one, the value `1` (`0001`) would become the value `2` (`0010`) because the digit in the fourth position from the right is moved to the third position and a `0` fills in where the `1` originally was. Hence, `0001` becomes `0010`, or the decimal value `2`. Using bitwise operators in JavaScript, you can complete the same kinds of operations. The following shows a single shift to the left, with decimal `3` becoming decimal `6`:

```
<html>
<head>
<title>Bitwise shift</title>
<script language="JavaScript">
var alpha = 3 << 1;
document.write(alpha);
</script>
</head>
<body bgcolor="palevioletred">
</body>
</html>
```

The output to the screen is `6`, but, for JavaScript, it simply shifts `0011` to `0110`. Four bit operations are shown, but JavaScript converts the values to 32-bit integers internally so that all floating-point numbers are converted to integers and are rounded down (for example, `3.9999` becomes `3`).

Using *typeof*

The `typeof` operator is unary, returning one of the following values:

- `number`
- `string`
- `boolean`
- `object`
- `function`
- `undefined`
- `null`

To use the operator, type the operator (`typeof`), a space and the operand, or place the operand in parentheses after the `typeof` operator. The following script shows the return of an array (`object`) and a Boolean value (`boolean`) using both methods of applying the operator:

```
<html>
<head>
<title>typeof Operator</title>
<script language="JavaScript">
var lots=new Array();
var whatTruth = 10 > 4;
var   kindOfData1= typeof lots;
var kindOfData2 = typeof(whatTruth);
var kindOfData=kindOfData1 + "<p>" + kindOfData2;
document.write(kindOfData);
</script>
</head>
<body bgcolor="wheat">
</body>
</html>
```

The *new, delete,* and *void* Operators

Of these last three operators discussed, `new` is the most commonly used. All objects must begin with a constructor preceded by the `new` operator. As seen previously, the `array` object begins with the `new` operator:

```
var family = new Array("Dad","Mom","Sue","Kris");
```

Likewise, other constructors for objects all use `new`.

The `delete` operator removes an object property or array element in a script. For example, the following would undefine the array element with the string value `Sue`:

```
var family = new Array("Dad","Mom","Sue","Kris");
delete family[2];
```

However, despite the operator's name, the element is not deleted; only the value is. The following script demonstrates what happens:

```
<html>
<head>
<title>Delete Element Value</title>
<script language="JavaScript">
var family = new Array("Dad","Mom","Sue","Kris");
delete family[2];
document.write(family[3] + "<p>"+family.length);
</script>
</head>
<body bgcolor="peru">
</body>
</html>
```

The length of the array is still four, and the last element is still `Kris`. However, the third array element (`element[2]`), while no longer `Sue`, still exists. The `delete` operator simply undefined it.

The final operator, `void`, is unary and operates on any literal or variable. Usually, you will see this operator as part of an `<A>` tag in an HTML script, such as here:

```
<a href="javascript:void(0) "onClick="scroll(500,0)">
```

The `void` operator suppresses the display of values from evaluated expressions. All the viewer sees is `javascript:void(0)` in the window in the lower-left corner when the mouse moves over the link instead of the full expression, including the URL.

Precedence

The order in which expressions are evaluated based on their operators is known as precedence. Multiplication and division occur before addition and subtraction, so any operands that are to be multiplied or divided occur before ones that are added and subtracted. Precedence can be reordered by placing expressions within parentheses. The innermost parentheses are evaluated first and work outward. So, if you want two numbers added before multiplication, place them in parentheses. The following two script excerpts show the difference results from different precedence order:

```
var alpha = 3 * 4 + 7 //alpha's value is 19 — 12 + 7
var beta = 3 * (4 + 7) // beta's value is 33 — 3 * 11
```

When all of the operators have the same precedence, the evaluations occur from left to right. Table 4.3 is a precedence chart, with the lowest ranks being executed before the higher ones.

Table 4.3. Precedence

Rank	Operators
1	`. [] ()`

Rank	Operators
2	`++ --` - (negation) `~ ! delete new typeof void`
3	`* / %`
4	`+` - (subtraction, addition, or concatenation)
5	`<< >> >>>` (bitwise shifts)
6	`< > <= >=`
7	`= = != = = = != =`
8	`&` (bitwise)
9	`^` (bitwise)
10	`|` (bitwise)
11	`& &`
12	`| |`
13	`?:` (ternary)
14	`=` All compound assignments (such as `+=`, `/=`, and `&=`)
15	`,`

Summary

Operators are truly understood and appreciated only with use and practice. Most debugging of JavaScript is a matter of looking to make sure that all of the characters that make up the bulk of operator symbols are the correct ones and that they are placed where they belong. Of course, memorizing which ones do what is important, but so many of them depend of the context of their use that only lots of practice and debugging work leads to optimum use.

In this chapter and previous chapters, operators were used in the context of legal JavaScript statements. The next chapter examines the different statements that have been used, plus others not yet used in examples. As you will see, doing them effectively requires using them in concert with the set of operators from this chapter. All of the actions in the expressions are controlled by the operators and, in the context of a statement, JavaScript performs different actions.

When a designer understands the JavaScript basics, he is in a position to begin thinking about how to use variables, operators, and data to communicate with the viewer. By recording what the user does in variables, the site can respond in ways limited only by the designer's imagination. For example, if the user moves the mouse over one position, that movement can be placed in the variable `alpha` simply by changing the value of `alpha` to record the movement. Likewise, if the user moves to another position, the movement can be recorded in a different variable, `beta`. Later in the book, you will see how all different types of input from the mouse and keyboard can be used in site design to individualize feedback to viewers based on their movement on the page.

Chapter 5. JavaScript Structures

CONTENTS>>

As seen in Chapter 4, "Using Operators and 4," operators are the building blocks of expressions. In some respects, expressions are the building blocks of statements; however, expressions themselves can be statements, so it would be tautological to say that expressions build statements.

A better way to understand statements can be found in the three basic structures in JavaScript and just about every other programming language. The three structures are as follows:

- Sequences
- Branches
- Loops (iterative)

All statements in JavaScript fall into one of those structural categories, with the possible exception of functions. Functions are encapsulated statements or compound statements, but the statements within a function are one of the three structures. So, a function, too, is made up of one or more of the basic structures.

Scripts and functions can use single or compound structures. A compound structure is simply a structure that uses two or more of the three basic structures. Most scripts do. For example, the following script uses all three structures:

```
<html>
<head>
<title>Compound Structure</title>
<script language="JavaScript">
var counter = 0; //Sequence
var endIt = 14;
while (counter < endIt) {        //Loop begins
      if (counter == 7) {             //Conditional begins
            alert("You are half way through");
            }                       //Conditional ends
      counter++;
}                               //Loop ends
var message="Counter value is now:" + counter;
document.write(message);
</script>
</head>
<body bgcolor="antiquewhite">
</body>
</html>
```

Absolutely no conflicts exist between structures, and one structure can be used in conjunction with or even inside (nested) another. For example, in the previous script, a conditional structure is nested inside a loop structure.

Getting back to what a statement is, just about any line of code in a JavaScript program is a statement. The keywords and combination of expressions, or even expressions themselves, can be statements. Some statements have a commandlike consequence (operations), while others define and redefine variables and objects. Still others perform transformations and calculations.

Statements in Sequential Structures

Sequential structures refer to the order in which the code is entered into a JavaScript program. In one sense, all code is sequential, but I am using the structure to differentiate it from statements that loop or conditionally execute lines of code.

Variable Declarations and Assignments

A simple statement that declares a variable using `var` is all that a variable declaration does. Typically, a variable is declared and assigned a value at the same time; however, technically, the two types of statements are different. For example, a perfectly legitimate variable declaration is this one:

```
var myFineVariable; //Declaration
```

Later in the script, the developer may assign a value to the declared variable. For example, she might write this, without having to use the `var` keyword:

```
myFineVariable = alpha + beta; //Assignment
```

The assignment statement assigns a value, and a declaration statement makes the variable part of the script. (Even though the declaration statement is optional in JavaScript, use it always. Among other things, you can avoid confusion that might arise between global and local variables.) The assignment of the variables `alpha` and `beta` *assumes* that in the sequence of the script, both `alpha` and `beta` have been declared and assigned values themselves. Hence this sequence must precede that assignment of those variables to another variable:

```
var alpha= 5;
var beta= " de Mayo";
```

Function Definitions

A second type of definition statement in JavaScript is the function definition. Unlike the optional `var` keyword, the `function` keyword is absolutely required in JavaScript. This form is used to define all functions:

```
function functionName(optional argument) {
      Statements
}
```

An argument is optional after the function name, but the parentheses are required. For example, the following script illustrates a function definition statement with no arguments:

```
<html>
<head>
<title>Function No Arguments</title>
<script language="JavaScript">
```

```
function hiThere( ) {
      document.write("Hi there!");
      }
hiThere( );
</script>
</head>
<body bgcolor="lavender">
</body>
</html>
```

In other cases, you probably will want to provide arguments because the parameters change. For example, the following script has a function that measures the cubic inches for a dog crate. The arguments are for *x, y,* and *z* dimensions representing width, height, and depth. By adding the optional arguments in a function, it's easy to use the same function to find different dimensions.

```
<html>
<head>
<title>Function With Arguments</title>
<script language="JavaScript">
function dogCrate(x,y,z) {
      var size = x * y * z;
var message="Your pup will have " + size;
message += " square inches in the crate.";
document.write(message);
      }
dogCrate(15,30,50);
</script>
</head>
<body bgcolor="tan">
</body>
</html>
```

The sequence of statements within the function follows the same rules as a sequence outside of a function. Within the curly braces, the multiple statement creates a *compound statement.*

Function Calls

In previous chapters, a common function call has been this:

```
alert("Some message");
```

Function calls in a script include both built-in and user-built functions. Many of the built-in functions might look not like functions, but like methods in objects. However, the `Math` object is a placeholder for functions and constants. For examples, the following script uses a function call as part of a variable assignment:

```
<html>
<head>
<title>Calling Function</title>
<script language="JavaScript">
var BidA="33";
var BidB="24";
var bestBid=Math.max(BidA,BidB);
var message ="The best bid is $" + bestBid;
document.write(message);
</script>
</head>
<body bgcolor="lightcyan">
</body>
</html>
```

Because most built-in functions will not return anything that you can use unless you put the return into a variable, a function call by itself won't help much. For example, this line:

```
Math.max(BidA,BidB);
```

is equivalent to typing

```
33;
```

Nothing happens. Function calls to `alert( )`, `prompt( )`, and `confirm( )`, plus user functions with an output statement, can create a return that can be displayed on the screen. Or, as in the previous script, the function call is assigned to a variable.

Increment/Decrement Statements

In examining operators in the previous chapter, both the increment (`++`) and decrement (`- -`) operators were discussed. Whenever they are used in a lines, such as this one, they constitute an increment/decrement statement:

```
addOne++;
- -dropOne;
```

Usually, you will find these statements as a part of a loop statement.

Conditional Structures

The "thinking" structure in JavaScript is found in the different types of conditional statements in the language. Used in concert with different types of comparative operators, conditional statements take the script on different routes, depending on what conditions have been met.

At the same time that JavaScript has a thinking structure, so should designers. The ability to fluently write your own scripts rather than cutting and pasting *someone else's design* frees you from that person's vision of a page or page component. Let JavaScript figure out what the user is doing, and provide the user with an interesting response from JavaScript rather than something that you don't understand but can only cut and paste.

The *if* Statement

When testing for a condition to execute one or more statements, the `if` statement is the most common to use. It has the following general format:

```
If (condition) {
      Conditional Statement(s)
}
```

The conditional statement is executed *only* if the condition resolves to a Boolean `true`. Otherwise, the script continues to the next line *after* the second curly brace.

Single or multiple conditions can be a part of the triggering condition. The following script contains a single condition that resolves as `false` so that the conditional statement is not executed.

```
<html>
<head>
<title>False Condition</title>
<script language="JavaScript">
var alpha="High";
var beta="Low";
var message="The condition is not met";
if(alpha > beta) {
      message="The condition is met";
      }
document.write(message);
</script>
</head>
<body bgcolor="mediumspringgreen">
</body>
</html>
```

The expression found to be `false` is the condition that the variable `alpha` is greater than the variable `beta`. Because `beta`’s value is `Low` and `alpha`’s value is `High`, and because letters higher in the alphabet are resolved to be greater than letters lower in the alphabet, the `false` Boolean value prevented the script from executing the conditional statement. When the condition is changed to this:

```
if(beta >alpha) {
```

the condition is found to be true, and the value of the variable message is changed to “The condition is met,” and that’s what appears on the screen.

Multiple statements (compound statements) may appear within the curly braces in an `if` statement, allowing several different events to occur. For example, the following example has three different statements when a condition is met in the `if` statement:

```
<html>
<head>
<title>Multiple Statements in Conditional</title>
```

```
<script language="JavaScript">
var alpha="Zebras";
var beta="Monkeys";
if(alpha > beta) {
//"Zebras" are greater than "Monkeys" because 'Z' is further up the
alphabet than 'M.'
var polite="Please enter your name:"
var yourName=prompt(polite);
alert("Hiya " + yourName);
}
</script>
</head>
<body bgcolor="beige">
</body>
</html>
```

The else Keyword

The limitation of the `if` statement by itself is that no alternative branch is made available for a false condition. So another keyword, `else`, had to be added as an alternative form of `if`. The following format uses *two sets* of curly braces:

```
if (condition) {
      Conditional statement(s)
} else {
      Different conditional statement(s)
}
```

For example, in the following example, a Boolean outcome forces a different branch (conditional statement) for a `true` or `false` value:

```
<html>
<head>
<title>If Else</title>
<script language="JavaScript">
var stillSmokin="cough";
var quitSmokin="freeAtLast";
if(stillSmokin > quitSmokin) {
      alert("You\'re going to die too soon fool!");
      } else {
      alert("Way to go Jack!");
}
</script>
</head>
<body bgcolor="whitesmoke">
</body>
</html>
```

In scripts with user input, such as forms or prompt functions, the `else` option provides a step for a second type of feedback. When the parser (interpreter) is going through the code line by line, the `else` statement is interpreted *only* if the first condition is `false`.

The *else if* Convention

Sometimes several options must be considered and several alternatives must be provided. The `else if` "statement" combines the `if` keyword and the `else` keyword into a conventionally used pair to create a unique statement. Combining `else` and `if` beyond a single `if` keyword differentiates it from the standard combination of `if` and `else`. Consider the following `else if` format:

```
if (condition1) {
      Conditional statement/s 1;
}
else if (condition2) {
      Conditional statement/s 2;
}
else {
      Conditional statement/s 3;
}
```

Because the `else if` "statement" is not a unique JavaScript word but rather is a programming convention, what is really happening is that the first `if` statement can be used with the first `else` statement. The `else` branch is to another `if` statement. Therefore, the last statement in an `else if` sequence is the lone `else` statement.

```
<html>
<head>
<title>else if Structure</title>
<script language="JavaScript">
var puppy=prompt("What kind of pup would you like?","");
var puppyLC=puppy.toLowerCase( );
if(puppyLC=="greater swiss mountain dog") {
       alert("Yes we have Swissies!");
       }
       else if(puppyLC=="great dane") {
             alert("Yes we have those big wonderful Great Danes!");
             }
       else if(puppyLC=="irish wolfhound") {
             alert("Yes we have the Gentle Giants!");
             }
       else {
             alert("Sorry we only have giant dogs.");
             puppy="information where to find that breed";
            }
var message="<p>Come get your <b>" + puppy;
message +="</b> at<h3>Goliath\'s Breeders</h3>";
document.write(message);
</script>
</head>
<body bgcolor="palegreen">
</body>
</html>
```

The final `else` statement is typically used as a residual category, one in which the `if` statements exhausted the categories provided in the series of `else if` combinations. It works like a "none of the above" selection in a multiple-choice quiz.

Using *switch, case,* and *break*

The series of `else if` combination statements makes multiple comparisons against a condition. JavaScript provides an alternative to the repeated checking conditions using the `switch` and `case` statements:

```
switch(expression) {
case alpha:
      Alpha statements execute
      break;      // skip the other cases if case alpha==expression
case beta:
      Beta statements execute
      break;      // skip the other cases if case beta==expression
default:     //if no matches execute this
      Tell user that nothing matches
}
```

To see how the `switch` and `case` keywords work together in a script, the next script takes a similar topic as was done with the `else if` statements. Using `switch` and `case` as statements, the `switch` statement includes what amounts to a *true condition* to be matched with the different cases. In most real-world applications of `switch`, the contents of the expression in the `switch` statement would be based on data from external input by a user.

If the `case` matches the expression in the `switch` statement, the statements in that `case` are executed. Then the parser moves down to the next line and into the next `case` statement. To prevent that from happening, one of the statements within each case should be `break`. Because the `break` statement is executed only if the `case` statement for that segment of the script is `true`, the only time that `break` will affect the parsing of the script is when the condition that is sought in the `switch` statement has been found. Thus, when `case` resolves as `true`, `break` moves the script execution out of the larger `switch` condition (beyond the closing curly brace) and on to the next line of JavaScript.

```
<html>
<head>
<script language ="JavaScript">
var puppy="Irish Wolfhound";
puppy=puppy.toLowerCase( );
var found;
switch(puppy) {
case 'great dane':
alert("Big Guy Breeders have Great Danes");
found="Big Guy Breeders phone: 555-9943";
break;
case 'irish wolfhound':
alert("Gentle Giant Breeders have Irish Wolfhounds");
found="Gentle Giant Breeders phone: 555-1912";
break;
case 'greater swiss mountain dog':
alert("The Swissy Center Breeders have Greater Swiss Mountain Dogs");
found="The Swissy Center Breeders phone: 555-5432";
break;
default:
```

```
alert("Contact the American Kennel Club for other breeds and
breeders.");
found="American Kennel Club: 555-8989";
}
var message="<p><p>Be sure to contact them as soon as possible";
message +="<h2>" + found + "</h2>"
document.write(message);
</script>
</head>
<body bgcolor="lightgreen">
</body>
</html>
```

NOTE

Using `break` *is sometimes associated with poor programming practices, and it generally should be avoided in conditional statements, especially for novices. However, the* `break` *keyword is a perfectly legitimate one and has useful applications that conform with good programming; using break with* `switch` *and* `case` *is a good example of the* `break` *keyword's appropriate use.*

Placing the `break` at the end of every case within a `switch` statement is optional, but doing so is good practice to save processing time and protect against errors. *Some* uses of `case` and `switch` might mitigate against using `break` (for example, you might have more than a single matching case and want to launch different actions from within a `switch` statement with more than a single case), but, *for the most part,* using `break` with `switch` and `case` is a good practice.

Conditional Shortcuts

Ternary conditional operators were discussed in Chapter 4. As a reminder, a ternary conditional can be substituted for a simple `if` / `else` statement. For example, both of the following scripts do the same thing, except that the ternary conditional is far more concise.

Ternary Shortcut

```
2 > 3 ? alert("It is true") : alert("Not true!");
```

Standard if/else Statement

```
if(2 > 3) {
      alert("It is true");
      } else {
      alert("Not true!");
}
```

You can save some coding time with the ternary operator conditional shortcut, and while it is perhaps not as clear as the standard `if` / `else` statement, once you get used to using the shortcut, you will find it helpful to get through a project quickly. The following script shows how the ternary shortcut appears in the context of a script:

```
<html>
```

```
<head>
<title>Conditional Shortcut</title>
<script language="JavaScript">
2 > 3 ? alert("It is true") : alert("Not true!");
</script>
</head>
<body>
</body>
</html>
```

Loops

Loops in JavaScript are similar to loops in C++ and Java and most other languages using loop structures. In this section, you will find explanations of the different types of loops in JavaScript and suggestions where they are typically used most effectively in a script. In previous chapters, several examples used loop structures to illustrate how to employ operators and variables. Now in this section, the loop structures themselves are the focal point of discussion.

The *for* Loop

One of the most used and familiar loops is the `for` loop. This loop iterates through a sequence of statements for a number of times determined by a condition. The condition can be a constant based on a numeric literal (a number) or a constant (that is, a math constant), or the loop can be variable depending on the count in the variable. The general format is shown here:

```
for(start value; termination condition; increment/decrement) {
      Statements
      }
```

The *start value* is the initial value of a counter variable. The first time through the loop, the counter value will be based on the start value. The *termination condition* is a test to determine whether the counter variable has met the condition that terminates the loop. The *increment/decrement* determines how much has been added or subtracted from the counter variable. A typical use for a loop is to examine characters in a string. The length of the string is used as the termination condition, and each character is based on its linear position in the string.

```
<html>
<head>
<title>For Loop</title>
<script language="JavaScript">
var found = "Email address is missing @ symbol.";
var emailAd=prompt("Please enter your email address:","");
for (var counter=0; counter <= emailAd.length; counter++) {
//The charAt(n) function looks at the character 'n' in the string
            var findAt=emailAd.charAt(counter);
                  if (findAt=="@") {
                              found="Email address has @ symbol";
                  }
}
document.write(found);
</script>
</head>
```

```
<body bgColor="powderblue">
</body>
</html>
```

Because the length of the string is a variable, the termination condition uses the length of the string rather than a literal value. In this particular example, all that the script is attempting to do is verify whether the user remembered to put in the "@" when she entered her email address.

The *for/in* Loop

A second format used with the `for` keyword in a loop is the `for` / `in` statement. When the `for` / `in` statement is used, the counter and termination are determined by the length of the object. The general format is shown here:

```
for (counter variable in object) {
      Statement
}
```

You do not need to know the number of properties in the object using `for` / `in` because the statement begins with 0 as the initial value of a counter variable and terminates the loop when all of the properties of the objects have been exhausted. For example, using an array object, the following loop begins with the first element of the array named `airplane` and keeps looping until no more elements are found in the array:

```
<html>
<head>
<title>For Loop</title>
<script language="JavaScript">
var airFlock="";
var airplane = new Array("Cessna","Piper","Maule","Mooney","Boeing");
for (var counter in airplane) {
      airFlock += airplane[counter] + "<br>";
}
document.write(airFlock);
</script>
</head>
<body bgColor="powderblue">
</body>
</html>
```

Because *variables are objects* in JavaScript, each character of a string variable is a property of the variable. Rewriting the script used to illustrate how a `for` loop works, the following `for` / `in` loop requires a simpler statement to arrive at the same results:

```
<html>
<head>
<title>Search For/In</title>
<script language="JavaScript">
var complete="You are missing the @ character in your email address.";
var emailAd = prompt("Enter your email address","");
for (var counter in emailAd) {
      if (emailAd[counter]=="@") {
            complete="You included your @ character.";
      }
}
document.write(complete);
</script>
</head>
<body bgColor="aliceblue">
</body>
</html>
```

Using the `for` / `in` loop in simple strings is just as effective as its use in other objects that contain properties.

The *while* Loop

The `while` loop begins with a termination condition and keeps looping until the termination condition is met. The counter variable initialization and the counter increment/decrement are handled within the context of the `while` statement (that is, within the curly braces), but they are not part of the initial statement itself. The general format for the `while` loop is shown here:

```
initial value declaration
while (termination condition) {
      statements
      increment/decrement statement
}
```

As long as the termination condition is not met, the statements are executed and the counter variable increases or decreases in value. The following example illustrates the counter variable decrementing in steps of 5:

```
<html>
<head>
<title>While Loop</title>
<script language="JavaScript">
var counter = 50;
var teamGroups="";
while(counter > 0) {
      teamGroups +="Team " + counter + "<br>";
      counter -= 5;
}
document.write(teamGroups);
</script>
</head>
<body bgColor="teal">
</body>
</html>
```

The output to the screen is as shown:

```
Team 50
Team 45
Team 40
Team 35
Team 30
Team 25
Team 20
Team 15
Team 10
Team 5
```

The fact that no Team 0 exists is important. As soon as the termination condition returned a Boolean `false`, the loop was immediately terminated and the script jumped over the statements within the loop and executed the next line. Had the termination condition been this, a Team 0 would have been included in the output:

```
while(counter >= 0) {
```

The *do/while* Loop

Unlike the `while` loop, the `do` / `while` loop *always* executes statements in the loop in the first iteration of the loop. Instead of the termination condition being at the top of the loop, it is at the bottom. The general format looks like the following:

```
do {
        statements
        counter increment/decrement
} while(termination condition)
```

The keyword `while` is *outside* the curly braces beginning after the `do` keyword. Because arrays are commonly used with loops, the following shows a `do` / `while` loop extracting the properties of an array:

```
<html>
<head>
<title>Do/While Loop</title>
<script language="JavaScript">
var bigCities= new Array("Beijing", "Tokyo", "Mexico City", "New
York", "Los
Angeles", "London", "Berlin", "Bloomfield")
var counter=0;
var metropolis="";
bigCities.sort( );
do {
      metropolis += bigCities[counter] + "<br>";
      counter++
} while (counter < bigCities.length)
document.write(metropolis);
</script>
</head>
<body bgColor="cornsilk">
</body>
</html>
```

The sorting statement, `bigCities.sort( )`, puts the array elements into alphabetical order before the array is placed in the loop. Then the loop iterates until the counter variable returns a Boolean `false` based on the length of the array. Because the elements have been arranged alphabetically, the output is arranged alphabetically, as the following shows:

```
Beijing
Berlin
Bloomfield
London
Los Angeles
Mexico City
New York
Tokyo
```

The *with* Statement

Like the ternary conditional operator, the `with` statement is a shortcut. Instead of having to list all of the properties of an object by repeating the basic object, you can state the bulk of the object in a `with` statement and then the properties within the context of the `with` statement. For example, take a typical form object. First you must state the object as follows:

```
document.formName...
```

Then you follow with the element names and values:

```
…elementName.value
```

Thus, your statement would be

```
document.formName,elementName.value
```

A typical form could include several different elements, such a first name, last name, address, city, state, ZIP code, Social Security number, and all other kinds of property details. Using the `with` statement, you can specify the object name once and then follow it with *all* of the properties and their values in this format:

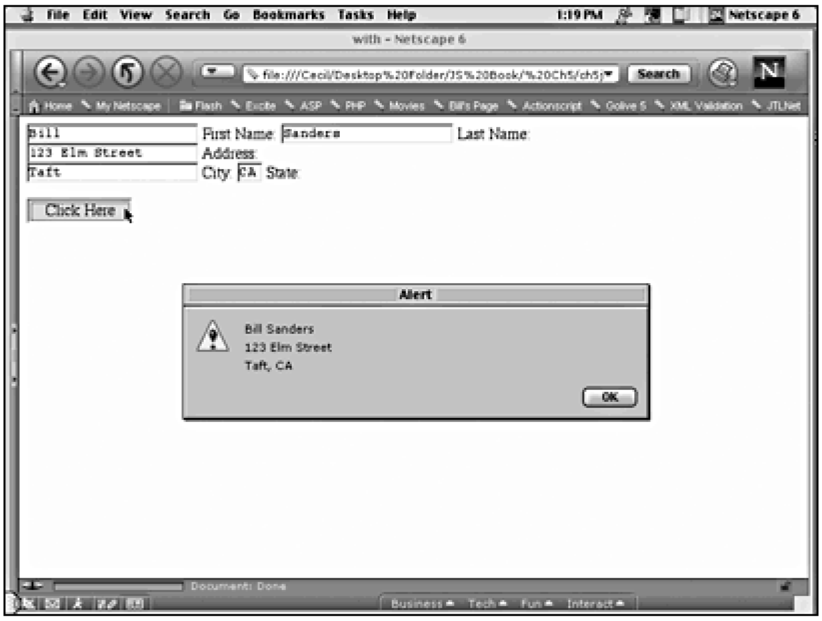
```
with (object) {
statements with properties only
}
```

The statements are typically property values. In the following simple example, the script uses a single `with` statement and then places the property values of the object in the statements within the `with` context:

```
<html>
<head>
<title>with</title>
<script language="JavaScript">
function showEm(){
      with (document.customer) {
      var alpha=fname.value;
      var beta=lname.value;
      var gamma=address.value;
      var delta=city.value;
      var epsilon=state.value;
}
var fullName=alpha + " " + beta + "\n";
var livesHere=gamma + "\n" + delta + ", " + epsilon;
alert(fullName + livesHere);
}
</script>
</head>
<body bgColor="cornsilk">
<form name="customer">
<input type=text name="fname"> First Name:
<input type=text name="lname"> Last Name:
<br>
<input type=text name="address"> Address:
<br>
<input type=text name="city"> City:
<input type=text name="state" size=2> State:
<p>
<input type=button value="Click Here" onClick="showEm()">
</body>
</html>
```

When and where to best use the `with` statement depends on the application, but, as can be seen from the example, it helps to clean up and simplify references to multiple properties in an object. Figure 5.1 shows the output in a browser.

Figure 5.1. When using forms containing multiple properties, consider using the `with` statement.



Bill First Name: Sanders Last Name:
123 Elm Street Address:
Taft City: CA State:

Bill Sanders
123 Elm Street
Taft, CA

The *label* and *continue* Statements and Nested Loops

The `label` statement does not inherently *go with* the `continue` statement but, like discussing `break` with `switch` and `case`, you might find it useful to see the statements used in a mutual context. Likewise, nested loops typically are written without either label or continue statements, but they serve as a useful structure to help explain how to effectively use continue.

For the most part, I don't use continue because, like the break statement, it can signal sloppy programming practices and poor planning. However, when used appropriately and in the right context, continue can be a valuable programming option. The statement jumps out of sequence in a loop structure, but, unlike break, which exits the loop, continue jumps to test the termination condition of the loop, effectively skipping the current iteration of statements within the loop.

Consider a program in which a baseball team is sequentially given jersey numbers *except* for the numbers of specially recognized players whose numbers have been retired. Within a loop, the continue statement can jump to the beginning of the loop when any of the retired numbers are found in the loop. Furthermore, you have more than a single team, and the second team has the same number of players and uses the same jersey numbers. The first loop (outer) keeps track of the teams, and the second loop (inner) keeps track of the players and jerseys that they will be getting. When one loop resides inside another loop, it's called a *nested loop.*

In JavaScript, labels are not statements, but rather identifiers. If you have ever programmed in Basic, in which line numbers or labels are used to reference a line of code, you know what labels are. They are places in the script where the program can branch if a statement tells it to do so. The format for a label is as follows:

```
label:
statements
```

In some respects, labels can be used like comments to help you organize your scripts, but they also can be used in conjunction with continue to send the program to execute the labeled portion of the script. Because the continue statement can be used only in loops, labeling the loops helps to control what the program will do. In the following script, the two loops are labeled team and jersey. Within the jersey loop is a conditional statement using continue that prevents the retired team numbers from being used. Note that the continue statement commands a jump to the beginning of the jersey loop, not the team loop. After you run the script, change the label next to continue from jersey to team.

```
<html>
<head>
<title>Using Continue and Labels</title>
<script language="JavaScript">
var teamJ="";
var teamMember=0;
team:
     for(var outCount=1;outCount<3;outCount++) {
            jersey:
                  for(var inCount=20;inCount<35;inCount++) {
                        if(inCount==22 || inCount==29 || inCount==30)
{
                              continue jersey;
                        }
```

```
                    if (teamMember==12) {
                    teamMember=0;
                    }
                                       teamMember++;
     teamJ += "Team" + outCount + "Member " + teamMember + " Jersey
Number " + inCount +
     "<br>";
     }
}
document.write(teamJ);
</script>
</head>
<body bgColor="mediumspringgreen">
</body>
</html>
```

The script output should look like the following:

```
   Team1 Member 1 Jersey Number 20
   Team1 Member 2 Jersey Number 21
   Team1 Member 3 Jersey Number 23
   Team1 Member 4 Jersey Number 24
   Team1 Member 5 Jersey Number 25
   Team1 Member 6 Jersey Number 26
   Team1 Member 7 Jersey Number 27
   Team1 Member 8 Jersey Number 28
   Team1 Member 9 Jersey Number 31
   Team1 Member 10 Jersey Number 32
   Team1 Member 11 Jersey Number 33
   Team1 Member 12 Jersey Number 34
   Team2 Member 1 Jersey Number 20
   Team2 Member 2 Jersey Number 21
```

It finishes with Member 12, and then starts over with Member 1.

Notice how all of the retired jersey numbers were omitted in the assignments for both teams. Now change this line:

```
   continue jersey;
```

to

```
   continue team;
```

When you run the program a second time, the output shows only the following four lines:

```
   Team1 Member 1 Jersey Number 20
   Team1 Member 2 Jersey Number 21
   Team2 Member 3 Jersey Number 20
   Team2 Member 4 Jersey Number 21
```

The reason that the second script produces only four lines in the browser window is that, as soon as the first retired number was detected, the program branched to the outer loop (team), incremented the value of the counter, and ended when the second reserved number was found because it had reached the termination condition. So, as you can see, depending on which label the `continue` statement branches to, very different outcomes are produced.

Summary

The three basic structures of sequence, branch, and loop are well represented in the rich assortment of statements in JavaScript. Knowing how and when to use the different structures and the set of statements that create the structures along with the operators and expressions makes up the heart of any language and certainly JavaScript. Throughout the remainder of the book, the set of statements discussed in this chapter and the operators from the previous chapter are used repeatedly. All of the objects and functions, other than those that are built into JavaScript, employ different combinations of statements and operators discussed up to this point. In the next chapter on functions, the bulk of the discussion is centered on how to use statements to create different functions using the operators and expressions discussed up to this point.

Chapter 6. Building and Calling Functions

CONTENTS>>

In discussing the different structures, objects, statements, operators, and expressions in JavaScript, you have seen several examples using functions. Chapter 3, "Dealing with Data and Variables," explained global and local variables and discussed how local variables are defined inside of function definitions. Chapter 1, "Jump-Starting JavaScript," showed how JavaScript in the deferred mode uses functions with event handlers to enable the designer to build a page that responds to the user's actions with input devices. This chapter focuses on the different uses and constructions for functions.

Methods and Functions

In previous and subsequent chapters, you will see references to methods and discussions about methods that make them sound a good deal like functions. To help clarify the difference between methods and functions, this section skips ahead a bit to a discussion of objects in JavaScript. (The next chapter discusses objects in detail.)

Objects are collections of properties arranged in a hierarchy. The highest level of objects in the context of JavaScript and an HTML page is the window. Everything in an HTML page is a property of the window object. When a function is the property of an object, it is called a *method.* However, technically speaking, because everything in an HTML page is a property of the window object, all functions are methods associated with the window property. When using most methods, you must specify the object with which they are associated. For example, the following script uses a method and function found in previous chapters:

```
<html>
<head>
<title>String Function/Method</title>
<script language="JavaScript">
var fullName = "Willie B. Goode";
var address="123 Polar Bear Drive";
address=address.toLowerCase();
window.document.write(fullName + "<br>" + address);
</script>
</head>
<body bgcolor="snow">
</body>
</html>
```

The output of the script is this:

```
Willie B. Goode
123 polar bear drive
```

The string variable `address` has a property that functions to change all of the characters in the string to lowercase. The function, `toLowerCase()`, is called a *method* because, in this case, it always is going to refer to the string object of which it is a property. The other string in the script, `fullName`, is not affected at all by the function because the function (method) is not used as a property of the `fullName` string variable.

In the same script, the familiar `write()` function is used to print the two strings to the screen. However, the full line is `window.document.write()`. As you can see, both `document` and `write()` are properties of the `window` object; hence, the `write()` function can be seen as a method. The reason that `write()` and many other built-in functions (such as `alert()` and `prompt()`) are considered functions instead of methods is that it is unnecessary to write the window object to fire off a function that is actually a property of the window object. For example, try the following little script, in which two built-in functions can clearly be seen as methods of the window object:

```
<html>
<head>
<title>Built-in Window methods</title>
<script language="JavaScript">
var watchThis=window.prompt("Enter your name","");
window.alert("See I told you " + watchThis + " that functions are
really methods.");
</script>
</head>
<body bgcolor="white">
</body>
</html>
```

While it should be clear that built-in functions are actually properties of—and, therefore, methods of—the window object, is the same true about user functions? Yes, it is. User functions are called methods when attached to an object. A script is worth a thousand words:

```
<html>
<head>
<title>User Window methods</title>
<script language="JavaScript">
function reallyMethod() {
      window.document.write("I\'m really a method!");
      }
window.reallyMethod();
</script>
</head>
<body bgcolor="white">
</body>
</html>
```

As you can see when you run the script, the user function is, in fact, a method in this case. The distinction between functions and methods, in a very strict sense, is artificial. However, in a practical sense, distinguishing between the two is helpful. If a method is a property of the window object and nothing else, it is called a *function.* If a function is a property of a lower-level specific object, it's called a *method.* In some contexts, you will find the terms used interchangeably. However, the important point is that all functions are *properties* of objects.

Creating Functions

Creating a function involves stating a function name and then adding a series of statements or other functions between curly braces. The general format is as follows:

```
   function functionName(optional arguments) {
   statements or other functions
   }
```

For example, the following function in the following script can be used to determine the number of 1 × 8 boards needed for a shed's walls. The two arguments in the function, `front` and `side`, are used as variables in two prompt functions.

```
<html>
<head>
<title>Shed Planner</title>
<script language="JavaScript">
function shedWalls(front,side) {
      var front=prompt("How many feet across the front?","");
      var side=prompt("How many feet along the side?","");
      var numFront=(front * 12 * 2)/8;
      var numSide=(side * 12 * 2)/8;
      var total=numFront+numSide;
      var message="You will need " + total + " one by eight-inch
boards for your
      shed walls."
      document.write(message);
      }
shedWalls();    //The function is called
</script>
</head>
<body bgcolor="palegoldenrod">
</body>
</html>
```

The `shedWalls()` function is launched from within the `<script>` container simply by placing the function in sequence after the definition.

It would have been simpler to leave out the function format altogether and simply have written the sequence of statements that were within the function itself. For functions to play the role that they were designed to play, they should be launched independently by an action taken by the user. This next section examines the events and event handlers used by JavaScript.

Firing Functions with Event Handlers

Chapter 10, “Event Handlers,” examines event handlers in detail. Here, however, you need to know how event handlers launch a JavaScript function. Chapter 10 explores the nuances of event handlers and the several different ones that are available in HTML and JavaScript.

Event Categories

Events can be divided into three main general categories:

- Keyboard and mouse events
- Load events
- Form-related events

Errors and window resizing can also be handled as events. Trapping errors can be used both in debugging and to keep a script from crashing. However, in this chapter, all the focus is on the main categories and how an event from any one works to fire a JavaScript function.

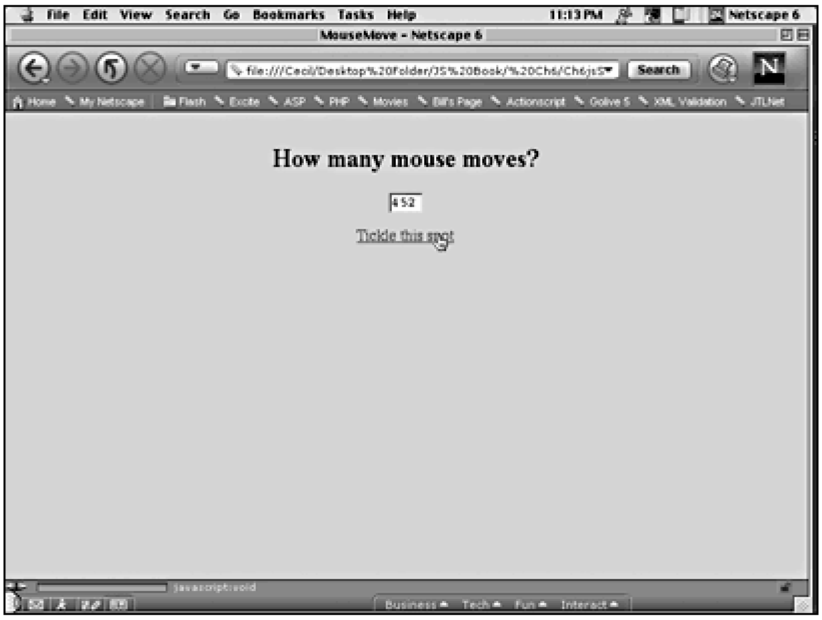
Mouse and Keyboard Events

Setting up functions to work with user-generated events from the keyboard or mouse is very different. As seen in previous chapters, capturing a mouse event is quite simple. Within a “hot” area on the page, such as a link or button object that detects mouse actions, you can launch a function in a script. For example, the following script keeps a running record of the mouse moving using the `onMouseMove` event handler from HTML. (NN6+ or IE5+ is required for this next script.)

```
<html>
<head>
<title>MouseMove</title>
<script language="JavaScript">
var recordIt=0
function moveIt() {
      recordIt += 1;
      document.viewer.spot.value=recordIt;
      }
</script>
</head>
<body bgcolor="pink">
<center>
<p>
<h2>How many mouse moves?</h2>
<form name="viewer">
<input type=text name="spot" size=3>
</form>
<a href=javascript:void onMouseMove="moveIt()">Tickle this spot</a>
</body>
</html>
```

To set up a dummy link, I used `javascript:void` as the reference point so that the link wouldn't try to jump elsewhere. (You can use the pound sign [`#`] to do pretty much the same thing.) The `onMouseMove` event handler from HTML fires the function named `moveIt()`. By incrementing a variable in the text box window each time the function is fired, you can get a better idea of how the event is recorded. If you change the script by substituting `onMouseOver` for `onMouseMove`, you will see that the text window indicates a new event only when the mouse is first over the hotspot. However, as long as any mouse movement is recorded with the `onMouseMove` event, the function is fired until the pointer is off the hotspot. Figure 6.1 shows the screen with the recorded movement as the pointer moves over "Tickle this spot" on the HTML page.

Figure 6.1. Any movement over the hotspot using `onMouseOver` fires the function to increment a variable stored in the text window.


File Edit View Search Go Bookmarks Tasks Help
11:13 PM
Netscape 6
MouseMove - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/%20Ch6/Ch6js5
Search
Home My Netscape Flash Excite ASP PHP Movies Bill's Page Actionscript Golive 5 XML Validation JTLNet
How many mouse moves?
452
Tickle this spot
javascript:void
Business Tech Fun Interact

Keyboard events require a whole different tactic of scripting and might even be considered a form-related event. They are discussed in detail in Chapter 10.

Load Events

When a page first appears on the screen, it "loads"; when it leaves, it "unloads." Both of these events can be captured as part of the `<body>` tag. Either an `onLoad` or an `onUnload` event handler within the `<body>` tag launches a function. The following shows how to launch a function as soon as a link to another page is clicked:

```
<html>
<head>
<title>unLoad</title>
<style>
h1 { color:red;
                  background-color:white
                  }
</style>
<script language="JavaScript">
function lastChance() {
      var message="Thanks for visiting and please come again.";
      alert(message);
      }
</script>
</head>
<body bgcolor="dimgray" onUnload="lastChance()">
<p>
<center>
<h1>Red Hot Page</h1>
</center>
</body>
</html>
```

The user is only indirectly involved with firing the function. Whereas a mouse click on a hotspot can fire a function repeatedly, a page loads and unloads only once. Be judicious in your use of `onUnload`. When people click a link to go somewhere else, they might not appreciate any delays.

Form-Related Events

The several form-related events are discussed in detail in Chapter 10, and here an example serves to illustrate how one form-related event works in firing a function. Unlike some of the other events that require a hotspot, the form-related events require a form. For example, the `onFocus` and `onUnfocus` events refer to placing the cursor into or removing it from an input text window within a form. When the writing cursor (I-beam) is placed into a text window, a "focus" event occurs and an `onFocus` event handler fires a function. The following script shows how a function can automatically fill a window:

```
<html>
<head>
<title>Focus This</title>
<script language="JavaScript">
function fillItIn() {
      //Substitute your own email address on the next line
      document.info.email.value="bill@sandlight.com";
      }
</script>
</head>
<body bgcolor="gainsboro">
```

```
<p>
<form name="info">
Please enter your email address:
<input type=text name="email" onFocus="fillItIn()">
</form>
</body>
</html>
```

As soon as you click the text window with your mouse pointer, the window should fill with the selected text. If you are using several text windows, pressing the Tab key on the keyboard will sequentially move text from one text window to the next. If you have an `onFocus` event handler for each text input window, you can provide a function to fill in each one.

The *return* Statement

The `return` statement is discussed here instead of in the last chapter because the statement can be used only as part of a function. The role of the `return` statement is to provide the value of the expressions within the function. But where does the function return the value to? For example, try out the following script:

```
<html>
<head>
<title>Return</title>
<script language="JavaScript">
function returnMessage() {
      var part1="Good ";
      var part2="Morning";
      var wholeThing=part1 + part2;
      return wholeThing;
      }
returnMessage();
</script>
</head>
<body bgcolor="mistyrose">
</html>
```

When you run the previous program, nothing appears on your screen other than having a nice, misty, rose-colored screen. The string value `"Good Morning"` *was returned* in the function. However, nothing in the function tells the script where to put the information in the function. Now, rewrite the script as follows:

```
<html>
<head>
<title>Return</title>
<script language="JavaScript">
function returnMessage() {
      var part1="Good ";
      var part2="Morning";
      var wholeThing=part1 + part2;
      return wholeThing;
      }
var putItOut=returnMessage();
document.write(putItOut);
</script>
```

```
</head>
<body bgcolor="mistyrose">
</html>
```

Now the script should provide you with a "Good Morning" greeting on the screen. By placing the function into a variable and directing that variable to be placed on the screen using the `document.write()` function, you have specified where you want the returned value to go. No doubt you might be wondering why you should use the `return` statement at all, and instead simply place a `document.write (wholeThing)` statement in the function to accomplish the same task. For optimizing the current script, the latter script might indeed be better; however, by using the `return` statement, you can treat the function as data in a variable or object.

In the previous script, try commenting out the `return` line like this:

```
//return wholeThing;
```

Save the script and run it again. On the second attempt, you should see "undefined" on the screen instead of "Good Morning." The value in the function is not returned, and so *nothing* is the value of the function. When nothing is placed into a variable, the return is always `undefined`. So, when you create a function in JavaScript, you need to remember to provide a `return` statement in the script if you plan to use the function as data in another expression.

Using Functions as Data

As illustrated in the previous section, functions can be treated as data. Like any other data, functions express some type of value, whether it be string, numeric, or Boolean. This next section considers two more methods of creating functions. The first method uses the `Function()` constructor, and the second uses function literals.

Using the *Function()* Constructor

The `Function()` constructor looks like the new object or array constructor. It has this general format:

```
var variableName=new Function("exp1", "exp2", "return exp3;");
```

Here, `exp1` is a necessary first expression, `exp2` is an optional second expression, and `exp3` is an expression made up of `exp1` and `exp2`. For example, the following script uses `item` as `exp1` and `tax` as `exp2` to generate a function that computes the total cost of an item, including tax:

```
<html>
<head>
<title>Function constructor</title>
<script language="JavaScript">
var total=new Function("item","tax", "return item +=tax");
document.write("Your bill is $" + total(14.43,.06));
</script>
```

```
</head>
<body bgcolor="thistle">
</html>
```

Unlike the `function` statement, the `Function()` constructor uses parentheses rather than curly braces, and all elements of the function are separated by commas. Also, the function has no name. Instead, the function-as-a-value is immediately placed into a named variable, and references to the function can be made through references to the variable name.

Using Function Literals

A newer version of the `Function()` constructor can be found in function literals. Function literals look more like function statements, in that they use curly braces. However, like the `Function()` constructor, they have no unique name of their own for purposes of reference. Their general format is as follows:

```
var variableName=new function(arg1,arg2) {return exp1};
```

Using the same parameter as in the `Function()` constructor example, you can see the similarities and differences in creating and using values derived from functions.

```
<html>
<head>
<title>Function Literal</title>
<script language="JavaScript">
var total=function(item,tax) {return item +=tax};
document.write("Your bill is $" + total(14.95,.06));
</script>
</head>
<body bgcolor="wheat">
</html>
```

Using functions as literal data provides a lot more flexibility in your scripts. Instead of invoking just a single function with an event handler, you can invoke a function that contains literals made up of other functions. Keep in mind when you begin using function literals in other functions that your scripts can exponentially increase in complexity and difficulty in debugging. However, rather than being a reason not to use function literals, this suggests keeping your script well organized so that you can see where everything belongs.

Properties in Functions

JavaScript functions are objects, and, as such, they contain properties. An important built-in property is `length`. The `length` property is a read-only one that returns the number of arguments that are *supposed to be* in a function. When you define a function, you can put in as many or as few, including zero, arguments in the function as you want. The number of arguments that you include becomes the value of the function’s length.

However, when you actually invoke a function, the number of arguments that you include may not match the number that you defined. The mismatch of defined and invoked arguments need not lead to your program crashing, but if the two are equal, you can be assured that the script is working as structured. A function property, `arguments`, has a length property as well. From `arguments.length`, you can find the actual number of arguments used in the function when it is invoked. By comparing the two values, you can debug your script. The following script uses a function with an unused but counted argument in the definition and three unused arguments when invoked. The output tells you whether they match.

```
<html>
<head>
<title>Function Properties</title>
<script language="JavaScript">
function bogus(xx) {
//The 'xx' is the argument
      var alpha=arguments.length;
      var beta=bogus.length;
      return alpha + " arguments, but should use " + beta + "
argument/s."
      //When the bogus.length run it returns 1 because it only has 1
argument.
      }
document.write("This function uses " + bogus(3,4,5));
</script>
</head>
<body bgcolor="moccasin">
</html>
```

As the name of the function implies, none of the arguments is used in the function. They are there only to illustrate how to access the properties and the fact that the arguments do exist as properties, whether they are used or not. Change the number of arguments in the `bogus()` function definition and in the statement that invokes the function to see the different feedback that you get on the screen. (Other browser-specific properties are available but, to minimize confusion, they are not discussed here.)

Methods in Functions

Only a single standard method object is available at this time for functions. Several other browser-specific methods are available but, as noted previously, this book focuses primarily on those that can be used with any browser. The method converts the output of a function to a string using the `toString()` method. It can be added to any function object. This format changes all numeric or Boolean values to strings:

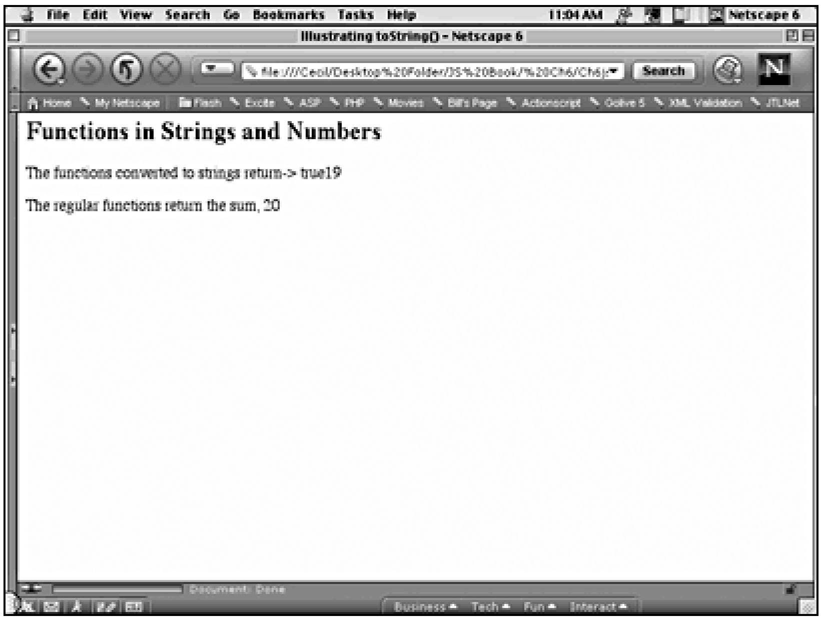
```
functionName.toString()
```

The following script provides an illustrative example of how the method works:

```
<html>
<head>
<title>Illustrating toString()</title>
<script language="JavaScript">
function alpha() {
      var hope = 12;
      var charity = 10;
      return hope > charity;
      }
function beta() {
       var now = 57;
       var then = 3;
       return now / then;
       }
var header="<h2>Functions in Strings and Numbers </h2>"
var indigo = alpha();
var denim=beta();
var boole = alpha().toString();
var lean = beta().toString();
var twoStrings= boole + lean;
var stringer ="The functions converted to strings return-> ";
var sumnum ="The regular functions return the sum, ";
var numnum=indigo + denim;
document.write(header + stringer + twoStrings + "<p>" + sumnum +
numnum);
</script>
</head>
<body bgcolor="lightyellow">
</body>
</html>
```

The purpose of including a Boolean value in the function `alpha` is to show that, expressed as a string, the value returns a `true` and, as a number, it returns the value `1`. When the two functions are converted to strings using the `toString()` method, the output is a concatenation of `true` and `19`. However, without the conversion, the value `20` is returned. Figure 6.2 shows the output on the screen.

Figure 6.2. The `toString()` method used with functions converts all numeric values to strings.



```
Functions in Strings and Numbers

The functions converted to strings return-> true19

The regular functions return the sum, 20
```

Summary

Functions are the bedrock of interactive JavaScript on the World Wide Web. Because functions, combined with event handlers, allow for a delayed response based on the user's action, they respond to an action and hence are interactive. So, virtually all events triggered by the user's action are potential sources for launching functions and engaging the user in ways that a static web page cannot do alone.

For the most part, functions are built as independent objects and are launched by a triggering event. However, they can be treated as data and can be placed into variables for use in other statements, including other functions as arguments or a part of other objects. As such, functions can be treated as modular building blocks in an object-oriented format for creating sophisticated but very clear scripts.

In developing your functions, use clear, concise code that you can reuse in other scripts. As you get better, your functions can be placed into libraries or an application program interface (API) and can be brought into your pages by an externally saved file with key JavaScript functions ready to use. All you have to do is call them into the new script, and a good hunk of your work will be done.

Chapter 7. Objects and Object Hierarchies

CONTENTS>>

In previous chapters, references have been made to objects, properties, and methods. I want to use this chapter to go into detail about the key role of objects in JavaScript and to provide a sense of the hierarchy of objects in HTML and JavaScript. In addition, this chapter introduces the concepts that you need to understand for object-oriented programming (OOP) in JavaScript. Thinking in terms of OOP and writing JavaScript code in OOP is the basis of programming excellence and program optimization.

Hierarchy of Objects in JavaScript

The key to understanding objects in JavaScript is to understand JavaScript's relationship to HTML and HTML's structure. One of the clearest and simplest structures in HTML is the form object. Table 7.1 shows the fundamental hierarchy of objects.

Table 7.1. Hierarchy of HTML Form

Window (object)
Document (property of window)
Form (property of document)
Element (property of form)
Element value (property of element)

In JavaScript, to reference the hierarchy, the dot (`.`) operator serves as a linking device for the different levels in the object hierarchy, with the highest level beginning on the left and working its way to the right. In JavaScript, a generic reference to the value of a form object would look like the following:

```
window.document.formName.elementName.value
```

Generally, the window object is assumed, so the reference begins with the document object. However, if you are using frames, a frame reference precedes the document reference. On the other end, `value` is the reference to any contents contained within the object to the left of it. Keeping in mind that an object is a collection of properties, many properties themselves are objects in their own right. Revisiting the hierarchy in Table 7.1, Table 7.2 shows how the object-property chain works.

Table 7.2. Object-Property Chain

Object	Properties
Window object	document, form, element, element value
Document object	form, element, element value
Form object	element, element value
Element object	element value
Element value	property of element

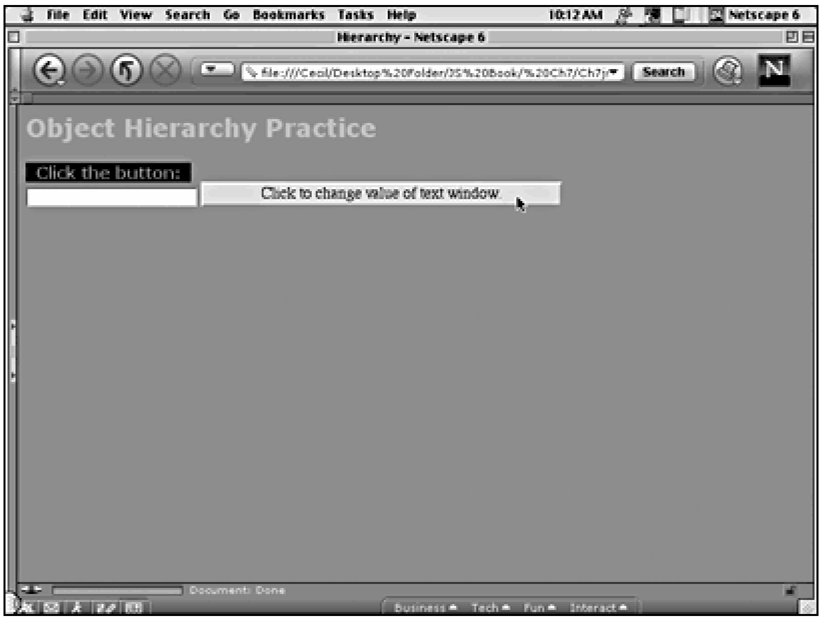
To make something actually work within the hierarchy, an example is in order. In this next example, the objects *are already defined* by HTML, and all that JavaScript will do is provide a literal value. However, as you will see in the script, more than a single object can have their values changed.

```
<html>
<head>
<title>Hierarchy</title>
<style type="text/css">
h2 {
      font-size: 18pt;
      color: #e8b002;
      font-family: verdana
}
.clickNote {
      font-size: 12pt;
      font-weight:bold;
      color: #fb503e;
      background-color:black;
      font-family: verdana
}
</style>
<script language="JavaScript">
function doIt() {
      window.document.petstore.puppy.value="Swissy";
      window.document.petstore.clicker.value="I\'ve  been clicked";
      }
</script>
</head>
```

```
<body bgcolor="#839063"
onLoad="javascript:window.document.petstore.reset()">
<h2>Object Hierarchy Practice</h2>
<span class="clickNote">&nbsp Click the button: &nbsp</span>
<form name="petstore">
<input type=text name="puppy">
<input type=button name="clicker" value="Click to  change value of
text window."
onClick="doIt()">
</form>
</body>
</html>
```

Figure 7.1 shows what appears as soon as the page is loaded *or* reloaded. An important JavaScript element in the script is the resetting of the text window, which is handled within the `<body>` tag.

Figure 7.1. The initial page shows a clear text window and a button with specific instructions.


File Edit View Search Go Bookmarks Tasks Help
10:12 AM
Netscape 6
Hierarchy - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/%20Ch7/Ch7
Search
Object Hierarchy Practice
Click the button:
Click to change value of text window
Document: Done
Business Tech Fun Interact

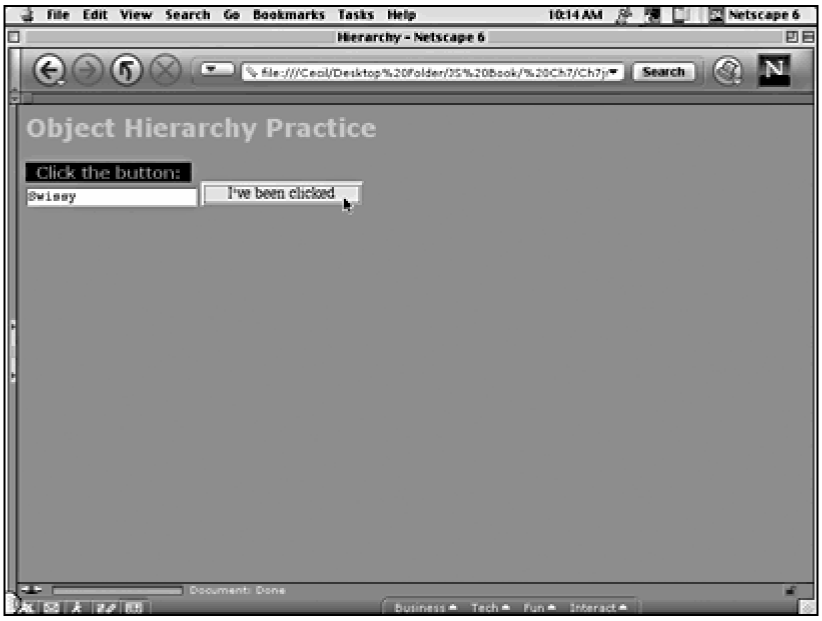
The style sheet simply sets up the colors and provides color values for the background and two of the fonts. The JavaScript function `doIt()` is made up of two object definitions that follow the hierarchy to a point where a value can be entered for a selected property. The text window was assigned no value in the HTML script, so the only value that it will have is the string literal `Swissy` that has been assigned to the object. Second, the button object has a value defined for it initially in the HTML script. However, to illustrate how JavaScript can change a property's existing value, it too is defined with a string literal: `I\'ve been clicked`. The hierarchy begins with `window`, but usually the top level of the hierarchy is superfluous and `window` was added only for the sake of showing the entire hierarchy.

A second bit of JavaScript outside of the `<script>` container is the line in the `<body>` container:

```
<body bgcolor="#839063" onLoad=
"javascript:window.document.petstore.reset()">
```

The JavaScript line is within the event handler `onLoad` using `javascript:` to specify the code that follows the JavaScript protocol. After the specifier, the `reset()` method is applied to the form object so that whenever the page loads, the undefined values in the text windows are cleared. Figure 7.2 shows the screen after the indicated button is pressed.

Figure 7.2. After the button has been pressed, both the values of the text window and button object are changed.



User-Defined Objects

Most of the objects discussed so far and those to be discussed in the rest of the book are defined in an HTML script. However, JavaScript provides a way that you can define your own objects with different properties. Each property can be assigned a value, just as with objects in HTML. The object constructor `Object()` is used to create user-defined objects using the following format:

```
var userObj = new Object()
```

After the object has been initialized, you can either add new objects to the existing object or add properties and their values. Consider an organization, the Acme Car Company, with the following five departments and two subdivisions:

Management

Research and Development

Finance

Production

Marketing

Sales

Fulfillment

To create a set of objects in JavaScript that reflects the organization, you will need to define an object on the highest level (Acme Car Company) and then the properties for each of the divisions and subdivisions. The initial definitions would be the following:

```
//Make the objects and properties
var acme= new Object();
      acme.management = new Object();
      acme.management = "Management coordinates everything.";
      acme.management.rand ="Research and Development works to
develop new and better
products and services.";
      acme.finance="Finance keeps track of the money.";
      acme.production = "Production makes the stuff."
      acme.marketing = new Object();
      acme.marketing = "Marketing develops strategies to target
markets for the
products."
      acme.marketing.sales = "Sales contacts potential clients and
sells the product."
      acme.fulfillment = "Fulfillment delivers sold products."
```

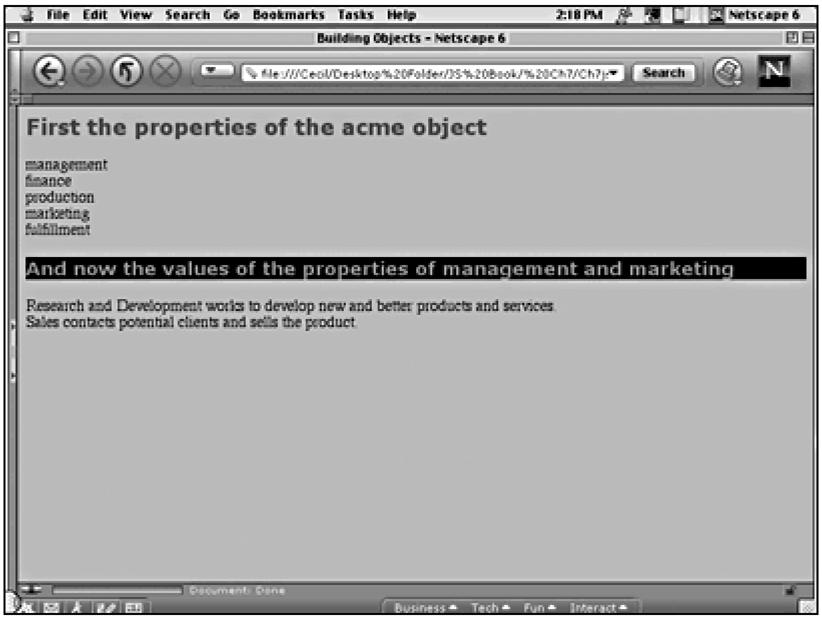
Because Management and Marketing both have subdivisions, they must be defined as objects as well as being properties of the `acme` object. Therefore, both Management and Marketing have their own `new Object()` constructor function that gives each the capacity to have their own properties. However, when `acme.management` and `acme.marketing` are defined as two new objects with the application of the `new Object()` constructor, they become independent objects even though they inherit the properties of the `acme` object.

To see how to use the objects' properties, the following script displays different elements within the `acme` object. First, using a `for` / `in` loop, the script pulls out all of the properties of the `acme` object simply to provide evidence that *it is an object.* Then, to show how to access the values of the object, a display variable adds on the current contents of the two subobjects of the `acme` object: `management` and `marketing`.

```
<html>
<head>
<title>Building Objects</title>
<style type="text/css">
h2 {
font-size:16pt;
color: #584095;
font-family: verdana
}
h3 {
font-size:14pt;
color: #6b8cc1;
background-color: black;
font-family: verdana
}
</style>
<script language="JavaScript">
var acme= new Object();
acme.management = "Management coordinates everything."; //Part of the
acme object.
acme.management = new Object(); //Creating new object
acme.management.rand ="Research and Development works to develop new
and better products
and
services.";
acme.finance="Finance keeps track of the money.";
acme.production = "Production makes the stuff.";
acme.marketing = "Marketing develops strategies to target markets for
the products.";
acme.marketing = new Object();
acme.marketing.sales = "Sales contacts potential clients and sells
the product.";
acme.fulfillment = "Fulfillment delivers sold products.";
var display="<h2>First the properties of the acme object</h2>";
for (var counter in acme) {
       display += counter + "<br>";
       }
display += "<h3> And now the values of the properties of management
and marketing</h3>";
display += acme.management.rand + "<br>";
display += acme.marketing.sales;
document.write(display);
</script>
</head>
<body bgcolor="#abc241">
</body>
</html>
```

Figure 7.3 shows what the output for the script is in a browser. Keep in mind that in the script, the `acme` object contains the five major divisions in the company: management, finance, production, marketing, and fulfillment. The `for` / `in` loop extracts only the properties of the `acme` object (the five divisions), not properties of the subobjects (the two subdivisions.)

Figure 7.3. The output is arranged as properties of the `acme` object and the values of the subobjects.



First the properties of the acme object

management
finance
production
marketing
fulfillment

And now the values of the properties of management and marketing

Research and Development works to develop new and better products and services
Sales contacts potential clients and sells the product

The two messages that appear at the bottom portion of the output are simply the values of the properties of the two subobjects (management and marketing). To access their values, the script simply placed them into a variable in the following format:

```
variableName = acme.management.rand;
variableName = acme.management.sales;
```

Note that *no value property* is specified, as has been seen in extracting values from other objects discussed previously in the book. For example, in the previous section of this chapter, a value was assigned to a text window object using this line:

```
window.document.petstore.puppy.value="Swissy";
```

The text window's name is puppy, so why not just define the value of puppy using the equal (`=`) operator without having to specify the `value` property? As an object, `text` has *two properties:* `onchange` and `value`. Without specifying which of the two properties you want to reference, JavaScript doesn't know what to do. However, in defining the properties in the previous script where user-definitions created the object, all of the properties of the `acme` object had their own values. For example, if you write this:

```
variableName= acme.finance;
document.write(varibleName);
```

your output would be

```
Finance keeps track of the money.
```

However, if you wrote this:

```
variableName= acme.marketing;
document.write(varibleName);
```

the output would be this:

```
[object Object]
```

In the first case, the script provides the complete object, `acme.finance`. In the second case, only a part of the object, as *ultimately defined* in the script, was presented, `acme.marketing` instead of `acme.marketing.sales`. To extract the *initial value* of the marketing object, you would need to put the value of `acme.marketing` into a variable before creating a second object that included sales, `acme.marketing.sales`. In the same way that a variable's value can change in a script, so can an object's. The last values assignment has the *last word* in determining what is read as a value in an object.

Built-in Objects and Their Properties

Several important built-in objects have properties that you need to understand for optimal use with JavaScript. This section examines two key objects: the screen and the window. The window properties examined here are specific to the window object and do not refer to virtually every other object that is a property of the window object.

Reading Screen Properties

When a browser opens an HTML page, it does so with the screen properties determined by the viewer's equipment. All screen properties are read-only and are static as far as the designer/developer is concerned. The screen object has five cross-browser properties:

```
screen.availHeight
screen.availWidth
screen.colorDepth
screen.height
screen.width
```

Each one of the screen property values can be assigned to a variable, another property's value, or, as in the next example, an array. When you have the screen data, you are in a position to make changes to your own script for maximizing what the viewer sees on his uniquely configured screen. When you try out the following script, do so on different browsers and different screen settings.

```
<html>
<head>
<title>Screen Properties</title>
<style type="text/css">
h2 {
      color: #ef3813;
      font-family:verdana;
      font-size:16pt;
      }
```

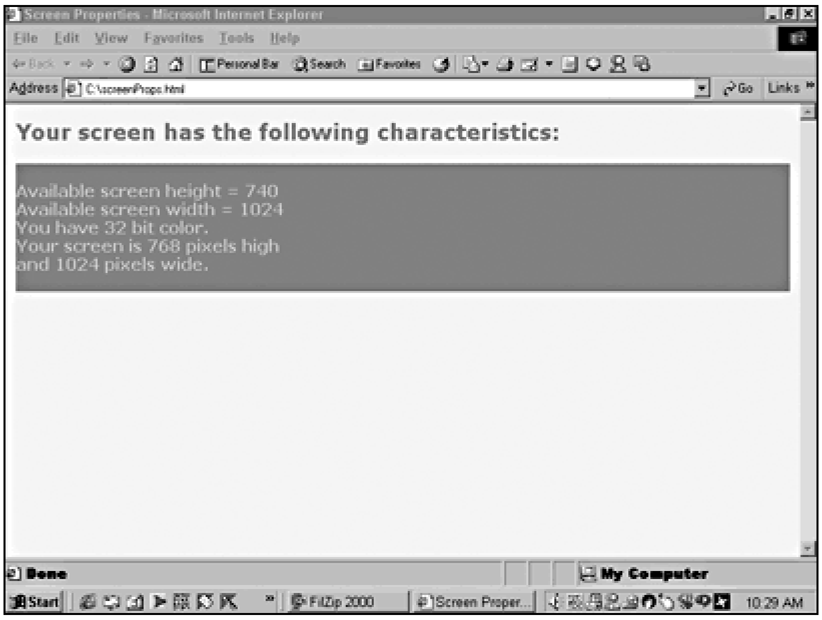
```
.outText {
      color: #e4c08e;
      font-family:verdana;
      font-size:12pt;
      background-color:#ef3813
      }
</style>
<script language="JavaScript">
function showScreen() {
      var display="<h2>Your screen has the following
characteristics:</h2><div
      class=outText><b><br>";
      var screenSee=new Array(5);
      screenSee[0] = "Available screen height = " +
screen.availHeight;
      screenSee[1] = "Available screen width = " + screen.availWidth;
      screenSee[2]=  "You have " + (screen.colorDepth) + " bit
color.";
      screenSee[3]= "Your screen is " + screen.height + " pixels
high";
      screenSee[4]= "and " + screen.width + " pixels wide.";
      for (var counter=0;counter< screenSee.length;counter++) {
            display += screenSee[counter] + "<br>";
            }
      display +="<br></b></div>"
      document.write(display);
      }
showScreen();
</script>
</head>
<body bgcolor="#fbf4eb">
</body>
</html>
```

When you run the program, you will see the setting that you have for your screen on your computer. I tested the script on an iMac and a Dell PC. The iMac returned the following output using Netscape Communicator 6.01:

```
Available screen height = 580
Available screen width = 800
You have 16 bit color.
Your screen is 600 pixels high
and 800 pixels wide.
```

The same script run on my PC has a different outcome because it has different screen settings. Figure 7.4 shows the output on the Dell using Internet Explorer.

Figure 7.4. The screen configuration on the PC shows a higher and wider screen and 32-bit color rather than 16-bit color.


Your screen has the following characteristics:

Available screen height = 740
Available screen width = 1024
You have 32 bit color.
Your screen is 768 pixels high
and 1024 pixels wide.

Because you cannot change the viewer's screen options, you might need to change the page that he sees. For example, using either multiple Cascading Style Sheets or images, your program can determine the color depth of the viewer's screen and decide which ones to use depending on the color depth. Designers who feel confined (or smothered) by the 216 web-safe palette might want to provide a wider variety of colors for viewers with more than 8-bit color. Of course, you run the risk that the screen's capacity to display color will be greater than the browser's, but some designers are willing to take this risk when they know that the screen is capable of greater color diversity.

Working with Window Properties

The properties that make up the window encompass the entire web page, so this section deals with only window-level properties. The following key properties are examined:

```
window.closed
window.defaultStatus
window.location
window.navigator
window.defaultStatus
window.name
window.history
```

To examine these properties, I used a frameset with three frames. I also developed a dummy page that slips into and out of one of the frames. To integrate the three frames, I created an external Cascading Style Sheet. First, create the style sheet and then create the frameset page.

winProp.css

```
h3 {
color:#c6d699;
background-color:#8d6ba8;
font-family:verdana;
font-size:16pt
}
.bod {
color:#731224;
font-family:verdana;
font-size:11pt;
font-weight:bold;
}
a:link {
color:#731224;
background-color:#c6d699
}
h2 {
color:#8d6ba8;
background-color:#c6d699;
font-size:14pt;
}
```

The frameset is arranged as a typical frame-delineated page. The top frame is named banner, the left column is named menu, and the larger right column is named main. The set provides three different pages that can be examined separately for the script and then viewed as a unit for the output.

winSet.htm

```
l
<html>
<frameset rows="35%,*" frameborder=0 framespacing=0 border=0>
<frame name="banner" src="frameA.html" border=0>
<frameset cols="30%,*" frameborder=0 framespacing=0 border=0>
<frame name="menu" src="frameB.html" border=0>
<frame name="main" src="frameC.html" border=0>
</frameset>
</frameset>
</html>
```

The top or banner frame goes through the navigator property. Note that in assigning a property value to the array, the reference is to `self.navigator`. Using `self` is simply a substitute for `window` and is a reminder of a self-referent.

frameA.html

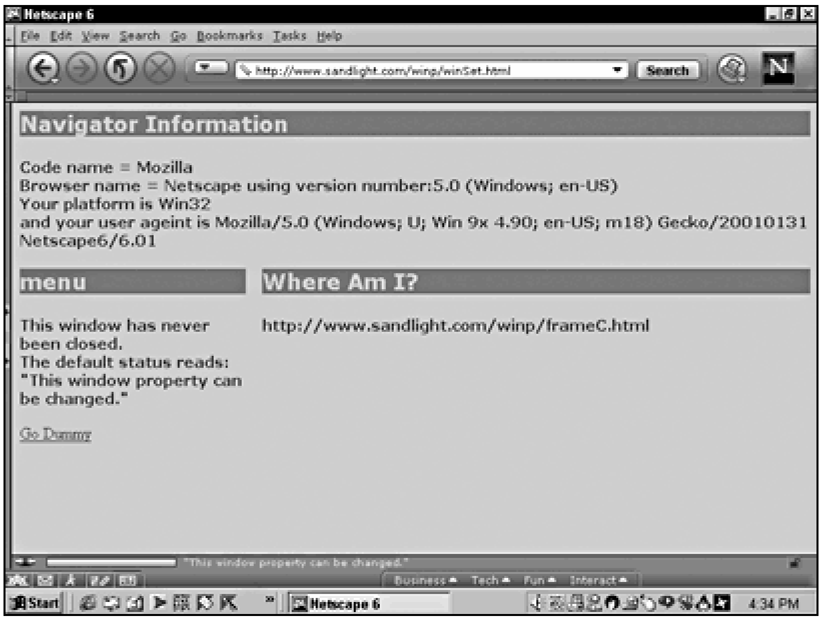
```
<html>
<head>
<title>Navigator Properties</title>
<link rel="stylesheet" href="winProp.css">
<script language="JavaScript">
var nav= new Array(5);
nav[0] ="Code name = " + self.navigator.appCodeName + "<br>";
nav[1] = "Browser name = " + self.navigator.appName + " ";
nav[2] = "using version number:" + self.navigator.appVersion+ "<br>";
nav[3] = "Your platform is " + self.navigator.platform+ "<br>";
nav[4] = "and your user agent is " + self.navigator.userAgent;
var display="<h3> Navigator Information</h3><div class=bod>"
for (var counter=0;counter < nav.length;counter++) {
      display += nav[counter];
      }
display +="</div>";
document.write(display);
</script>
```

```
</head>
<body bgcolor="#c6d699">
</body>
</html>
```

Depending on your system and browser, you can expect a different output from this frame. Try it out on different browsers on your system or, if you have more than one system, try it out on the different systems. The information available about your system and browser using JavaScript can be summarized by the output in the top frame. Other browser-specific properties of the Navigator object are available but are not listed here due to incompatibility problems.

In the second page, the window properties of `window.closed`, `window.name`, and `window.defaultStatus` are used. Note that the page itself can be swapped back and forth with the dummy page, but the `window.closed` status remains the same, as is evidenced by the message on the frame. Also look at Figure 7.5 to see where in the lower portion of the browser the message has changed due to changes in `window.defaultStatus`.

Figure 7.5. Information about a web page can be captured by JavaScript to optimize the page design and utility.


Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://www.sandlight.com/winp/winSet.html
Search

Navigator Information

Code name = Mozilla
Browser name = Netscape using version number:5.0 (Windows; en-US)
Your platform is Win32
and your user ageint is Mozilla/5.0 (Windows; U; Win 9x 4.90; en-US; m18) Gecko/20010131 Netscape6/6.01

menu

This window has never been closed.
The default status reads:
"This window property can be changed."

Go Dummy

Where Am I?

http://www.sandlight.com/winp/frameC.html

"This window property can be changed."

frameB.html

```
<html>
<head>
<title>Window Properties</title>
<link rel="stylesheet" href="winProp.css">
<script language="JavaScript">
var display = "<h3>" + self.name + "</h3>";
display +="<div class='bod'>";
var gone=" ";
var alpha=self.closed;
if (alpha==false) {
      var gone=" never ";
}
display += "This window has" + gone + " been closed.<br>";
```

```
//Change the message in the lower left corner
self.defaultStatus="\"This window property can be changed.\"";
display += "The default status reads: " + self.defaultStatus +
"<br></div>";
document.write(display);
</script>
</head>
<body bgcolor="#c6d699"">
<p>
<a href="dummy.html">Go Dummy </a>
</body>
</html>
```

The last frame is filled with a single window property, `window.location`, which identifies where your current page's URL is. It is a simple enough property, but it's one that could be passed to another variable for future reference.

frameC.html

```
<html>
<head>
<title>Window Properties</title>
<link rel="stylesheet" href="winProp.css">
<script language="JavaScript">
var display="<h3> Where Am I? </h3><div class='bod'>";
display += window.location;
document.write(display);
</script>
</head>
<body bgcolor="#c6d699"">
</body>
</html>
```

Rounding out the dummy page's purpose is nothing more than showing that the initial page that is swapped into and out of the "menu" frame does not have the effect of placing the initial page in a "closed" status. To provide a useful job for the dummy page, the link back to the original page in the menu frame is executed using a method from `window.history` to illustrate how it is used (see Figure 7.5).

dummy.html

```
<html>
<head>
<link rel="stylesheet" href="winProp.css">
</head>
<body bgcolor="#976d2c">
<h2>Dummy Page</h2>
<a href="javascript:window.history.back()">Back</a>
</body >
</html>
```

Key Built-in Object Methods

This section examines built-in object methods in some of the key objects in JavaScript. The three selected objects, window, string, and date, contain some of the most heavily used methods. As you might recall from the discussion of functions in Chapter 6, "Building and Calling Functions," a method is essentially a function attached to an object or property of an object. In looking at the basic methods of the following objects, you should be able to cope with the bulk of methods used regularly in JavaScript.

The Window Methods

At the time of this writing, JavaScript window object has 18 cross-browser methods. Some of the methods, such as `alert()` and `prompt()`, have been used in illustrations extensively, and I will not spend time going over them again. Because the window object has so many methods, rather than attempting a large script using all of them, several smaller scripts will be used to illustrate categories of window methods.

In the previous section of this chapter, and in previous chapters in this book, the window object was assumed, so, instead of having to write both the object and the method, only the methods were stated in the scripts. For example, instead of writing this:

```
window.alert("I don\'t need window to alert.")
```

you can write this:

```
alert("I don\'t need window to alert.")
```

Likewise, a window method can be prefaced by `self`. For the most part, though, you do not have to use the window object name. This first script shows how to use two methods in JavaScript, `setInterval()` and `setTimeout()`. In showing how to use the methods, the window object is placed in the script to remind you that the method belongs to the window object.

Timing Methods

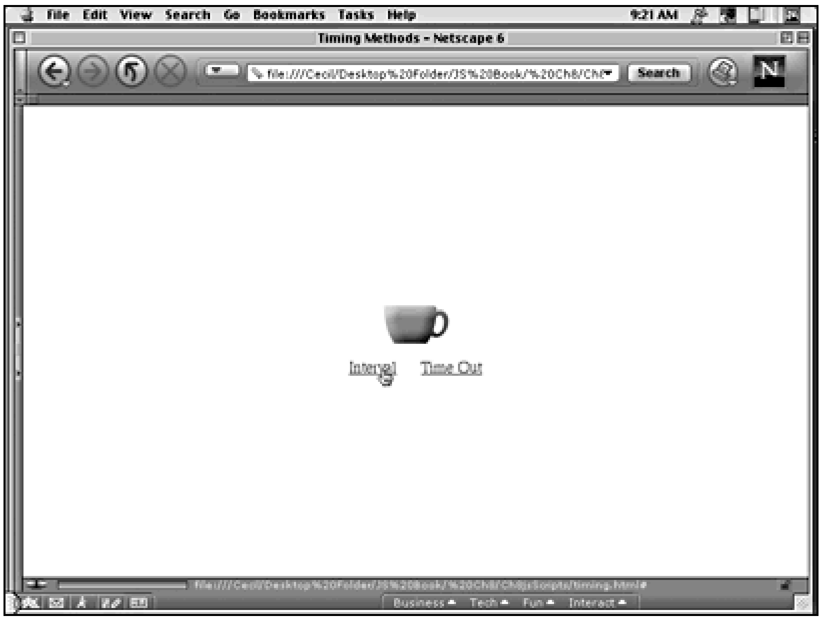
The first window methods to examine are ones involved in timing actions. The four methods are as follows:

```
setInterval(script,ms)
setTimeout(script,ms)
clearInterval(Id)
clearTimeout(Id)
```

The `setInterval()` method repeats a script action every so many milliseconds, initiating the script after the specified number of milliseconds. The `setTimeout()` method works the same as `setInterval()`, except that it does not repeat the script. Both `clearInterval()` and `clearTimeout()` cancel the actions initiated by the setting methods. In the following script, a coffee cup appears on the screen.

When the left hotspot is clicked, 3 seconds (3,000 milliseconds) elapse and the cup disappears. When the right hotspot is clicked, 5 seconds elapse and then the coffee cup reappears. (See Figure 7.6)

Figure 7.6. The `setInterval()` and `setTimeout()` methods add a delay before launching a script.



Two functions named `interval()` and `timeout()` do all the work. In the `interval()` function, a code replaces the coffee cup with a white image that has the effect of removing the cup from the screen. In defining the function, the object and method are placed into a global variable named `alpha`. The `alpha` variable is used in the `timeout()` function to identify the `setInterval()` method that the developer wants to turn off.

timing.html

```
<html>
<head>
<title>Timing Methods</title>
<script language="JavaScript">
var cupUp=new Image();
cupUp.src="cupaJava.jpg";
var cupDown=new Image();
cupDown.src="hideJava.jpg";
var alpha; //This variable needs to be global
function interval() {
alpha=window.setInterval("document.cup.src=cupDown.src",3000);
}
function timeOut() {
window.setTimeout("document.cup.src=cupUp.src",5000);
window.clearInterval(alpha);
}
</script>
</head>
<body>
<table cellpadding="0" cellspacing="0" border="0" width="100%"
height="100%"
align="Center">
    <tr valign="Middle">
       <td align="Center"><img name="cup" src="cupaJava.jpg"
alt="java" width="64"
       height="38"><p>
                  <a href="#" onClick="interval()">Interval</a>
                  &nbsp;&nbsp;&nbsp;&nbsp;
                  <a href="#" onClick="timeOut()">Time Out</a>
</td>
```

```
   </tr>
</table>
</body>
</html>
```

To get a good idea of what the `setInterval()` method does, comment out the line that turns off the interval in the `timeout()` function:

```
//window.clearInterval(alpha);
```

By doing so, the interval that makes the coffee cup disappear keeps repeating itself. The cup will keep disappearing after you hit the "Time Out" hotspot as the interval kicks in repeatedly.

Opening, Sizing, and Positioning Windows

This next set of methods includes methods to open and close windows, resize them, and move them. Each of the following window methods is employed in the next example scripts:

```
open()
close()
moveBy()
moveTo()
resizeBy()
resizeTo()
```

The `open()` method has access to most parameters, including `height`, `location`, `menubar`, `resizable`, `scrollbars`, `status`, `toolbar`, and `width`. You can use as many or few of these options as you want. Usually, designers have a certain look that they want for a window that is opened from an existing one and tend to turn off everything—a Boolean `0`, `no`, or `false` (as in `scrollbars=false`). The `close()` method is always self-referent with a page not part of a frameset. To close an opened window, the window itself must contain the `close()` method. Within a frameset, however, you can address the frame and window to be closed from another page in the frameset. The other methods are either relative (`By`) or absolute (`To`) for movement or resizing. One important difference between browsers can be found in using the `moveBy()` method. In *Netscape Navigator 4* and later, the window stops at the edges of the screen unless `UniversalBrowserWrite` privileges are invoked; however, Internet Explorer allows a window to be walked right off the screen. (See the note at the end of the script to see how Netscape allows offscreen windows.)

StandardWin.html

```
<html>
<head>
<title>Window Control</title>
<script language="JavaScript">
function GetStanWin() {

open("sampWin.html","sampWin1","toolbar=no,width=200,height=150")
      //width=640,height=460 recommended design size
      }
function moveRel() {
      moveBy(20,20)
      }
function moveAbs() {
      moveTo(20,20)
      }
function shrink() {
      resizeBy(15,10)
      }
function shrivel() {
      resizeTo(400,300);
      }
</script>
</head>
<body bgcolor="white">
<center><p>
<h2>Window Control</h2>
<form>
<input type="button" name="openBut" value="Click to Open New Window"
onclick="GetStanWin()"><p>

<input type="button" name="mvbyBut" value="Click to move relative."
onclick="moveRel()"><p>

<input type="button" name="mvtoBut" value="Click to move absolute."
onclick="moveAbs()"><p>

<input type="button" name="shrinkBut" value="Click to change relative
size." onclick="shrink()
"><p>
<input type="button" name="shrivelBut" value="Click to change
absolute size." onclick="shrivel()
"><p>
</center>
</form>
</body>
</html>
```

NOTE

Netscape handles security by using different requests for privileges. One privilege is `UniversalBrowserWrite`*, which allows, among other things, moving windows offscreen.*

To invoke the privilege, use the following line:

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserW
rite");
```

The next little sample window has a single method in it, `close()`. Note that no reference is provided the `close()` method. Any window with a `close()` method gets closed, so be careful where you put it.

sampWin.html

```
<html>
<head>
<title>Sample Window</title>
```

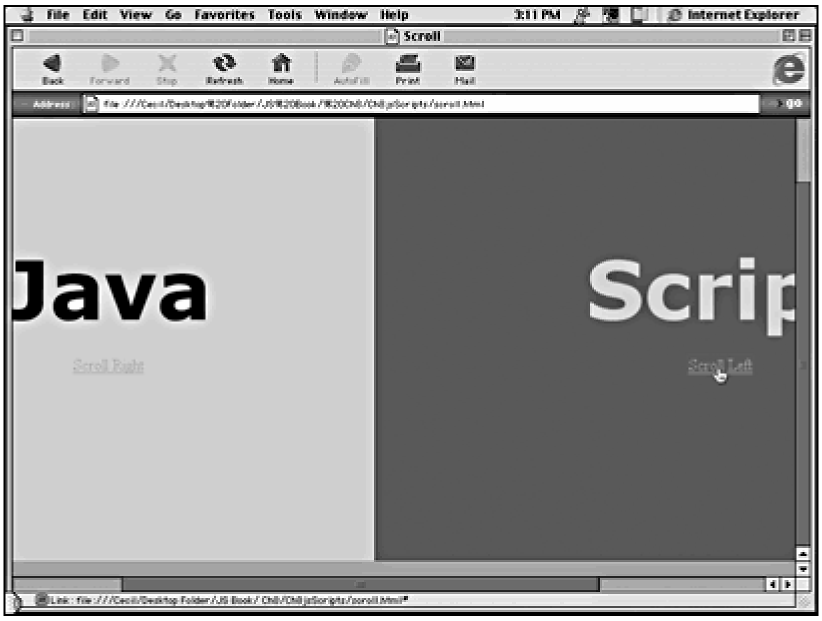
```
<script language="JavaScript">
function shutIt() {
        close();
        }
</script>
</head>
<body bgcolor="palegoldenrod">
<h4>This is a sample window.</h4>
<a href="#" onClick="shutIt()">Close</a>
</body>
</html>
```

Scrolling Methods

This section examines the last group of window methods available for design. Vertical control in an HTML page can be simplified by using anchors that will jump to different vertical positions in a page. However, some of the most interesting designs use a web page's horizontal real estate. Using the different scroll methods in JavaScript, you can create web pages that make full use of the horizontal plane in a web page. In designing pages for monitors with smaller screens, the scroll methods provide a way to have larger pages that are managed horizontally and vertically by scrolling options. Of the three scroll methods available, only two are being discussed. The `window.scroll()` method has been deprecated and replaced by `window.scrollTo()` in JavaScript 1.2. The `window.scrollBy()` method is a relative version of `window.scroll()`.

However, in experimenting with different browsers on different platforms, I found that IE5 on the Macintosh worked with the scrolling method the best, moving either left or right. Both of the major browsers on Windows Me and Windows 2000 had problems scrolling to the right, and the NN 4.7 and NN 6.01 on the Macintosh had similar problems in a movement to the right. Figure 7.7 shows the window scrolling back to the original position on the left.

Figure 7.7. A wide table creates a wide window for scrolling to the left and right.


File Edit View Go Favorites Tools Window Help
3:11 PM
Internet Explorer
Scroll
Back Forward Stop Refresh Home AutoFill Print Mail
Address
go
Java
Scrip
Scroll Right
Scroll Left

scroll.html

```
<html>
<head>
<title>Scroll</title>
<style type="text/css">
.jLeft {
      font-size: 60pt;
      font-family: verdana;
      color:black;
      font-weight: bold;
}
a {
color:#e39102;
}
.jRight {
      font-size:60pt;
      font-family: verdana;
      color: #e2d155;
      font-weight: bold;
}
</style>
<script language="JavaScript">
function goRight() {
     scroll(600,50);
     }
function goLeft() {
            scrollBy(-120,0);
     }
</script>
</head>
<body bgcolor="#e39102">
<table border=0 cellspacing=0 bgcolor="black" width=1200 height=500>
<tr>
<td align=center valign=middle bgcolor="#e2d155"><span
class=jLeft>Java</span><p><a
href="#" onClick="goRight()"> Scroll Right</a></td>
<td align=center valign=middle bgcolor="#983803"><span
class=jRight>Script</span><p><a
href="#" onClick="goLeft()"> Scroll Left</a></td>
</tr>
</table>
</body>
```

Summary

This chapter has taken the first major step in introducing the concepts of objects in JavaScript. As was seen, objects exist in a hierarchy in JavaScript, with the window object *always* being at the top of the heap. Everything else is a property of the window object in one way or another. Objects can be created by the designer/developer to add a needed element to a script, and these objects may have their own properties and values. Likewise, many of the built-in objects have their own properties that provide useful data for fine-tuning and customizing a script to address different configurations that viewers may have.

As you will see in the next chapter, objects are driven by their methods—functions associated with a particular object or one of an object's properties. By generating scripts that take advantage of the objects' properties and methods in JavaScript, you can create object-oriented programs. The advantage of object-oriented scripts is that they optimize the script's execution and open up a whole different and important way of organizing and thinking about the use of JavaScript.

Part II: Using JavaScript with Web Pages

Chapter 8. JavaScript Object-Oriented Programming and the Document Object Model

CONTENTS>>

Up to this point, all discussion about objects has been in terms of building them or addressing different object properties and methods. In some ways, properties act like variables in that they can be assigned values, and the values assigned to them can change. Likewise, you saw that methods are functions associated with objects and their properties. More importantly, though, many disparate pieces of code can be brought together in an object that contains all the information that variables provide and the actions inherent in functions. By conceiving of an object as a collection of *named* properties with *values* that can be *any type of data,* you can see them as building block structures in a script. Keeping in mind that functions can be a type of data, they are called *methods* when they are the value of a property. Perhaps it might help to think of objects as *coded modules* that can help organize your scripts.

Because JavaScript has objects, you need to think of programming with objects. Because I assume that most readers of this book come from a designing background rather than programming background, you will not be disappointed to find out that object-oriented programming (OOP) in JavaScript is different from OOP in C++ or Java. Rather than entering the fray about whether JavaScript is really an OOP language, I am going to treat JavaScript as a *type of OOP language.* In the same way that the Democrats and Republicans are two different political parties but still political parties, JavaScript is a type of object-oriented programming language, even though it might not be like other OOP languages. In the next section, all comments are directed at how JavaScript is an OOP language and what that means to someone writing OOP scripts.

Object-Oriented Programming in JavaScript

One way to think of JavaScript objects is as *associative arrays.* From our discussion of arrays and using the `for` / `in` statement to pull information from objects, you might have already suspected as much. To demonstrate this fact, the following script builds an object, assigns data to it, and then pulls it apart *as an array*:

```
<html>
<head>
<title>Associative Array</title>
<script language="JavaScript">
var tree = new Object();
tree.size="Big and taller than Uncle Jim.";
tree.shade="It keeps us cool in the summer";
tree.fall="We make giant leaf pumpkins."
var display =tree["size"] + "<br>";
display +=tree["shade"] + "<br>"
display +=tree["fall"]
document.write(display);
</script>
</head>
<body bgcolor="wheat" >
</body>
</html>
```

As you can see from the previous script, all of the properties of the `tree` object are essentially elements in the `tree` array. This is an associative array, and you can think of JavaScript objects as associative objects.

In working with functions, you can create objects made up of the function. These functions are called "constructor functions." Then when you create an object using the new constructor from the constructor function, your new object has the properties in the function. This following example uses a constructor function to set up a new object called `Bloomfield`. All of the `Bloomfield` properties inherit the properties of the `Mall` object.

```
<html>
<head>
<title>Constructing Method</title>
<script language="JavaScript">
//Constructor Function
function Mall(stName,shop,product) {
      this.stName=stName;
      this.shop=shop;
      this.product=product;
      }
var Bloomfield= new Mall("Bloomy","Nuts and Lightning Bolts","Organic
Electricity")
var display= Bloomfield.stName + "<br>";
display += Bloomfield.shop + "<br>";
display += Bloomfield.product;
```

```
document.write(display);
</script>
</head>
<body bgcolor="mintcream" >
</body>
</html>
```

As you will see in the output to the screen, all of the `Mall` object's properties were inherited by the `Bloomfield` object. (It might help to think of methods as *function properties* rather than simply functions attached to an object. Then you can better understand that a function *is indeed* a property.)

The Prototype Concept

Much of the difference between JavaScript and languages like Java is that OOP in Java is based on a different type of class than in JavaScript. In traditional OOP, a class is a *set* and any object is an instance of that class or set. However, in JavaScript, the concept of class revolves around the concept of a *prototype.* Unlike the set concept of class that treats an instance of an object to be a member of the class, the prototype concept treats the named object with all of the properties that all members of the class have. For example, consider the class called `Mall`.

A `Mall` object has several characteristics expected in all malls. The following lists what just about any mall has:

- A name (Bloomfield Mall)
- Shops (butcher, baker, and skateboard shop)
- A number of shops (15)
- Customers (number or individual names)
- Employees (number or individual names)
- Roles (clerk, manager, security, shoplifter)

In JavaScript, the concept of a class as applied to `Mall` suggests that `Mall` has all of the properties that malls typically have. Hence, you could have the following properties in dealing with the class `Mall`:

```
Mall.name
Mall.shop
Mall.shopNumber
Mall.customer
Mall.employee
Mall.roles
```

To see the difference between a variable and an object, the following script defines a `Mall` object and a `Mall` variable. The defined variable has been commented out using double slashes (`//`) because the `Mall` variable will not work as a defined object.

```
<html>
<head>
<title>Variable and Object</title>
<script language="JavaScript">
var Mall= new Object();
```

```
Mall.shops="Vinny's Gun and Medical Supply Shop";
//var Mall;
//Mall.shops="Louie's Used Golf Club Emporium and Discount Analysis
Clinic";
document.write(Mall.shops);
</script>
</head>
<body bgcolor="mintcream" >
</body>
</html>
```

If you remove the double slashes and comment out the original `Mall` object from the script, you will find that it will not work. The reason is because the variable defined in the second instance did not create an object. Instead, it created a variable—and, while even variables have certain properties (such as length), they cannot be assigned properties.

The *String* and *Date* Objects

Two important built-in objects have yet to be discussed, the `String` and `Date` objects. Each of these objects has important properties, methods, and parameters to understand. The `Date` object is fairly specialized as far as design is concerned, but it can be used in many interesting ways other than to display date and time. More significant for the designer is the `String` object and how it can be used to format data display output.

Using the String *Object*

The `String` object has a multitude of methods, but, at this point, only key selected ones are going to be discussed. At the same time, you can begin to look at a number of ways to format strings as objects and learn how you can use the built-in properties to control what you see on the screen. For example, the following script uses a string object employing *five methods* simultaneously:

```
<html>
<head>
<title>String Objects and Methods</title>
<script language="JavaScript">
var myWord = new String("It is impolite to wink!");
myWord=myWord.substring(18,22).fontsize(7).italics().fontcolor("red")
.blink();
document.write(myWord);
</script>
</head>
<body bgColor="peachpuff">
</body>
</html>
```

Notice that after the string object was defined using the `String()` constructor, it was possible to write this statement with five methods attached to the string object itself:

```
myWord=myWord.substring(18,22).fontsize(7).italics().fontcolor("red")
.blink();
```

By the end of the script, the string object `myWord` has properties of a substring, font size, italics style, a red font color, *and* a blinking font. (Don't ever use blinking fonts in real designs. They cause *style cancer*—besides, they work only in Netscape Navigator.) As you can see, using a string as an object, you can control what strings will look like on your page.

String methods can be divided into three categories, as shown in Table 8.1.

Table 8.1. Types of String Methods

Tag Methods	Action Methods	RegExp Methods
`Anchor()`	`charAt()`	`match()`
`Big()`	`charCode()`	`replace()`
`Blink()` (only in NN)	`concat()`	`search()`
`Bold()`	`indexOf()`	`split()`
`Fixed()`	`lastIndexOf()`	
`Fontcolor()`	`slice()`	
`Fontsize()`	`substring()`	
`Italics()`	`substr()`	
`Link()`	`toLowerCase()`	
`Small()`	`toUpperCase()`	
`Strike()`		
`Sub()`		
`Sup()`		

Tag Methods

The tag methods associated with strings are in reference to the associated tags in HTML. For example, the `bold()` method is associated with the `<b>` tag for creating bold fonts. Likewise, `fontcolor()` and `fontsize()` are associated with the `<Font>` tag. For designers, the message here is, "Be careful." In some ways, the string tag methods could be nearing extinction. Because of the power and flexibility of Cascading Style Sheets (CSS), many of the tags, such as `<Font>`, are being deprecated. If you get too accustomed to using the tag methods in JavaScript, you might find yourself riding a dinosaur. Take a look at Chapter 12, "Dynamic HTML," and see what you can do with CSS with JavaScript. In the meantime, you can see how many built-in methods you can add to a string.

Action Methods

I use the term "action methods" to delineate those methods that transform string composition or find out information about strings that do not use regular expressions. The action methods are the core ones in dealing with strings. The most important ones for design are substring and character identification.

- `substring(begin,end)` Enters the beginning and ending numeric positions of a part of the string to extract.
- `charAt(n)` Enters the value of the position of a character in a string.
- `charCodeAt(n)` Enters the value of the position of the character in a string to find the Unicode (ASCII) value of the character at position *n.*

- `indexOf(substr)` Enters a string fragment of the string to be searched. This returns the position of the first character of the string. The `lastIndexOf()` does the same thing but begins at the end of the string.
- `toUpperCase()/toLowerCase()` Transforms a string to all uppercase or all lowercase characters.

The following script provides an example of how all but the index methods can be used:

```
<html>
<head>
<title>Action Method</title>
<script language="JavaScript">
var quote= new String("She showed all of the emotions from A to B.");
var dex=quote.toLowerCase().substring(0,2);
var terity = quote.toUpperCase().charAt(22);
dex=dex.concat(terity);
document.write(dex);
</script>
</head>
<body bgColor="lavenderblush">
</body>
</html>
```

Regular Expression Methods

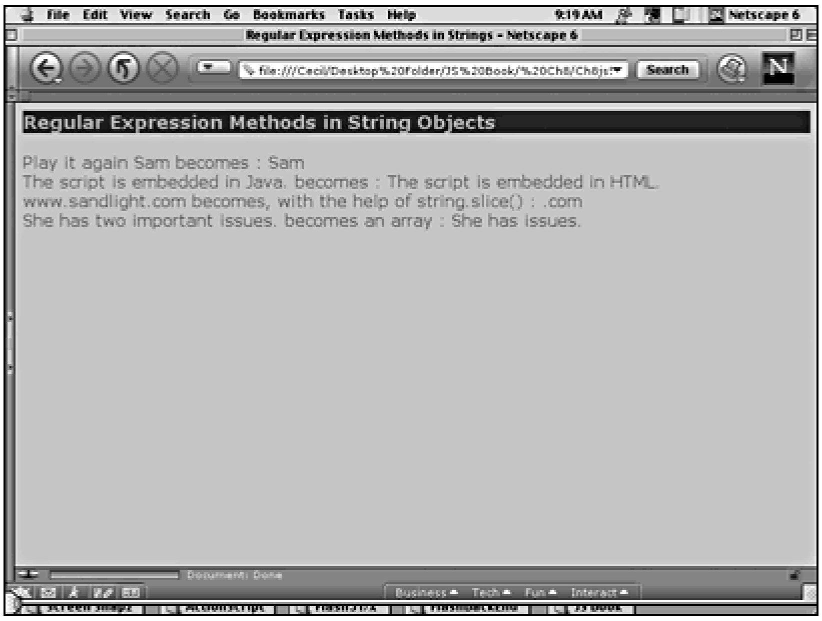
The last type of string methods to examine are those using regular expressions. How much you would actually use these methods depends on your page design, but regular expressions have great utility with strings. You might want to review the discussion of regular expressions in Chapter 3, "Dealing with Data and Variables," or take a look at the discussion of CGI and Perl in Chapter 16, "CGI and Perl," where many of the regular expression formats are discussed in greater detail.

The four regular expression methods for the string object are the following:

- `match(regexp)` This returns the substring in the regular expression or a `null` value.
- `replace(regexp,substitute)` The regular expression is replaced by the substitute string.
- `search(regexp)` This finds the starting position in the string of the regular expression.
- `split(regexp)` This method works with both strings and regular expressions (JavaScript 1.2). The string is split at the regular expression (possibly in more than one place), the terms in the regular expression are discarded, and the string is turned into an array with each element demarcated by the position of the discarded substring.

The following script demonstrates how each works. Note how the `String.search(exp)` method is used in a `String.slice()` method to find the term in the regular expression. The output from the script can be seen in Figure 8.1.

Figure 8.1. The regular expression methods associated with the string object provide a powerful addition to the tools that you can use in JavaScript formatting.


File Edit View Search Go Bookmarks Tasks Help
Regular Expression Methods in Strings - Netscape 6
Search

Regular Expression Methods in String Objects

Play it again Sam becomes : Sam
The script is embedded in Java. becomes : The script is embedded in HTML.
www.sandlight.com becomes, with the help of string.slice() : .com
She has two important issues. becomes an array : She has issues.

```
<html>
<head>
<style type="text/css">
body {
font-family: verdana;
background-color:#e6c303;
}
h3 {
color:#6dc303;
background-color:#1a291f;
}
</style>
<title>Regular Expression Methods in Strings</title>
<script language="JavaScript">
//string.match()
var matchThis = "Play it again Sam";
var stringer1=matchThis.match(/sam/i);//string.replace()
var replaceThis = "The script is embedded in Java.";
var stringer2= replaceThis.replace(/Java/g,"HTML");

//string.search()
var searchThis="www.sandlight.com";
var stringer3=searchThis.slice(searchThis.search(/.com/));

//string.split()
var splitThis = "She has two important issues."
var stringer4 = splitThis.split(/two important/i);

var display = "<h3>Regular Expression Methods in String Objects</h3>";
display += matchThis + " becomes : " + stringer1 + "<br>"
display += replaceThis + " becomes : " + stringer2 + "<br>"
display += searchThis + " becomes, with the help of string.slice() :
" + stringer3
+ "<br>"
```

```
display += splitThis + " becomes an array : " + stringer4[0]+ " " +
stringer4[1] +
"<br>"
var showOff=new String(display);
document.write(showOff.fontcolor("#972126"));
</script>
</head>
<body>
</body>
</html>
```

The different regular expression methods in the string object can save some time locating positions in strings. Instead of having to set up loops to search for a substring or position, think about using the regular expression methods.

Using the *Date* Object

The `Date` object has 40 methods and 9 arguments to keep you entertained for some time. However, many of the methods might not be used too often (such as the ones dealing with universal time), and most of the methods are either getting or setting the same thing. Nevertheless, you should be acquainted with both methods and arguments of the `Date` object. Table 8.2 provides a summary.

Table 8.2. Date Object Characteristics

Arguments	
`Milliseconds`	Number of milliseconds from January 1, 1970.
`Datestring`	Specified date and time (time is optional) in string format.
`Year`	Four-digit year (such as 2002).
`Month`	Months in numbers from 0 to 11, with January being 0.
`Day`	Day of the month expressed from 1 to 31. (Who knows why days begin with 1 and everything else begins 0?)
`Hours`	Hours expressed from 0 to 23.
`Minutes`	Minutes expressed from 0 to 59.
`Seconds`	Seconds expressed from 0 to 59.
`Ms`	Milliseconds expressed from 0 to 999.
Methods	Most of these return values are defined by the date that is input into a Date object, even if it isn't the current date/time.
`getDate()`	Returns the day of the month.
`getDay()`	Returns the day of the week.
`getFullYear()`	Returns the current year of the `Date` object.
`getHours()`	Returns the hours.
`getMilliseconds()`	Returns the milliseconds from 1/1/1970 to the date specified in the object.
`getMinutes()`	Returns minutes.
`getMonth()`	Returns the month as an integer.
`getSeconds()`	Returns the seconds.
`getTime()`	Returns the current time in milliseconds.
`getTimezoneOffset()`	Returns the time zone difference between where you live

	and UTC or GMT time.
GetUTCDate()	Returns the day of the month in UTC.
GetUTCDay()	Returns the day of the week in UTC.
getUTCFullYear()	Returns the current year in UTC.
getUTCHours()	Returns the hours in UTC.
getUTCMilliseconds()	Returns milliseconds in current UTC from 1/1/1970.
getUTCMinutes()	Returns minutes in UTC.
getUTCMonth()	Returns the month as an integer in UTC.
getUTCSeconds()	Returns the seconds in UTC.
Methods	
getYear()	Returns the year field, deprecated (getFullYear()).
setDate()	Sets the day of the month.
setFullYear()	Sets the year.
setHours()	Sets the hours.
setMilliseconds()	Sets milliseconds.
setMinutes()	Sets minutes.
setMonth()	Sets the month as an integer.
setSeconds()	Sets the seconds.
setTime()	Sets the current time in milliseconds.
setUTCDate()	Sets the day of the month in UTC.
setUTCFullYear()	Sets the year in UTC.
setUTCHours()	Sets the hours in UTC.
setUTCMilliseconds()	Sets the milliseconds field in UTC.
setUTCMinutes()	Sets minutes in UTC.
setUTCMonth()	Sets the month as an integer in UTC.
setUTCSeconds()	Sets the seconds in UTC.
setYear()	Sets the year, deprecated (setFullYear()).
toGMTString()	Converts a date to a string using UTC or GMT.
toLocaleString()	Converts date and time.
toUTCString()	Converts to a string using UTC.
valueOf()	Converts to milliseconds.

The date object is easy enough to use even with a dizzying array of options. However, for the most part, the date object is used to compare a past or present date with the current date or simply to display the date on the screen. Interesting world clocks can be made using the UTC (Coordinated Universal Time) methods as well as other uses for timing events, and it should not be forgotten when putting a date in your pages. The following script provides an example of how the date object might be employed to keep you on track for Valentines Day.

```
<html>
<head>
<script language="JavaScript">
var NowDate = new Date();
var DateNow= NowDate.getDate();
var MonthNow = NowDate.getMonth();
//To find Valentines Day you need 1 for the month and 14 for the day
//Remember months begin with 0 so February is 1 and days of the month
```

```
//begin with 1. Also be sure to get a hardcopy card!
if (DateNow== 14 && MonthNow == 1) {
        alert("Happy Valentines Day!")
        } else {
        alert ("Please wait until February 14 you romantic fool!")
}
</script>
</head>
<body bgcolor="pink">
</body>
</html>
```

Why Object-Oriented Programming?

This very short introduction to OOP has been an encouragement to begin thinking about JavaScript elements in terms of objects, their properties, and their methods. For most work in JavaScript, the scripts are short; while good programming organization is important for even short scripts, OOP is not essential for short scripts. However, as you start working with more people on a web project using JavaScript and the scripts become longer, OOP becomes more important. Because OOP encourages modular program units, the units can be shared with others on a programming team and can be reused, which means that you do not have to reinvent the wheel every time you sit down to work on your script.

As a web page designer, the concept of modular and reusable design elements is more obvious and intuitive. Object-oriented programming is the same. If you can write a complex piece of code as a module, the next time that same code is required, you can either make small changes to the module (such as different argument values) or use the same module elsewhere in the script. By doing so, the time spent on the original code design pays off in the long run.

The Document Object Model

In the JavaScript document object model (DOM), the primary *document* is an HTML page. As noted, the Window object, which includes frames, is at the top of the web browser hierarchy. However, the `Document` object contains the properties whose information is used by JavaScript. To strip away the mystery of what a DOM actually is, think of it as the `Document` object and all of its properties, including methods. Statements such as `document.write()` constitute the object (`document`) and a property/method (`write()`) that make up part of the model. To oversimplify a bit, you can say that the JavaScript DOM is the sum total of the `Document` object's properties and methods, including the arrays automatically generated on an HTML page, and the manner in which these objects are accessed by JavaScript.

Document Properties

In looking at the `Document` properties, you might experience some sense of *déjà vu* from the section on the `String` object earlier in this chapter. You will see some similarities, but, for the most part, the properties (not including methods) that make up the DOM are unique to the `Document` object. Table 8.3 shows the properties of the `Document` object.

Table 8.3. Document Properties

Property Name	Associated Property
AlinkColor	The active color of a link when selected.
Anchors[array]	Each anchor on an HTML page is an element of an array.
Applets[array]	Each applet on an HTML page is an array element.
BgColor	Document's background color.
Cookie	Text files that can be read and written by JavaScript.
Domain	Security property that can be used with two or more web servers to reduce restrictions on interactions between web pages.
Embeds[array] (also works = plugins Plug-ins [array])	Each embedded object is an array element of the embed[] array. and .SWF files are examples of embedded objects. (See Chapter 18, "Flash ActionScript and JavaScript," for a discussion of Flash-embedded .SWF files and JavaScript.)
FgColor	The text color.
Forms[array]	Each form on an HTML page is an array element, and each object within a form is an element of the form element. (See Chapter 11, "Making Forms Perform," for a full discussion of forms and JavaScript).
Images[array]	Each image placed on an HTML page is an element of the images[] array. (See later on in this chapter for a full discussion of the images[] array.)
LastModified	The last modification date, in string format.
LinkColor	The first color of a link before it has been visited. (This defaults to blue.)
Links[array]	Each link is an element in an array when it appears on a document.
Location	The URL property, now specified as URL. (See the URL entry, later in this table.)
Referrer	Previous page that had a link to the current page.
Title	The document title.
URL	New version of the location property specifying the URL of the loaded page.
VlinkColor	The color of a visited link.

The following script shows some of the document properties at work. The form object is an array, and the JavaScript function addresses the sole text object as an element of the form array.

```
<html>
<head>
<script language="JavaScript">
function message(hue) {
      document.forms[0].elements[5].value=hue;
      }
</script>
<title>Dynamic Color Change</title>
</head>
```

```
<body>
<H2> Big Font </h2>
<form>
<input type=button value="Red" onClick="document.bgColor='red'"
onMouseDown="message('red')">
<input type=button value="Green" onClick="document.bgColor='green'"
onMouseOver="message('green')">
<input type=button value="Blue" onClick="document.bgColor='blue'"
onMouseMove="message('blue')"><br>
<input type=button value="Font Color to Yellow"
onClick="document.fgColor='yellow'"> <br>
<input type=button value="Font Color to Firebrick"
onClick="document.fgColor='firebrick'"> <p>
<input type=text>
</form>
</body>
</html>
```

Other important properties of the `Document` object are the array, form, and image objects. Forms are discussed in detail in Chapter 11, and the array object was examined in Chapter 3. One important document array object that does bear closer scrutiny here is the image property as an object.

Image Objects

The image object is one of the most interesting in HTML and JavaScript. When you place images sequentially on an HTML page using the `<IMG>` tag, you place the image into an array. You do not declare an array, but you do create one simply by placing images on the screen. The array develops as follows:

```
document.images[0]
document.images[1]
document.images[2]
```

In the HTML page, the same images would appear as follows:

```
<img src = "graphicA.gif">
<img src = "graphicB.jpg">
<img src = "photoA.jpg">
```

One of the properties of the image object that can be changed dynamically with JavaScript is the `src` value. The following script uses single-pixel GIFs that have been resized to form vertical bars. To create a single-pixel GIF, just open the graphic program that you use for creating GIF images (such as Photoshop or Fireworks), and make the drawing screen 1 pixel by 1 pixel; then use the magnifying tool to make it big enough to work in. (All of the graphics are on the book's web site.) Color in the work area with one each of the following hexadecimal values:

#3a4c4f	#ce6c56	#859eab	#ffeb89
#abc5a8	#8c3227	#debe71	

Save each single-pixel GIF with the names c1.gif through c7.gif. The following script shows where each graphic is placed and shows the JavaScript that allows you to address the images array by an array address from 0 to 6. The script addresses the image as part of the `Document` object like this:

```
document.images[0-6].src=imageName.src
```

The `imageName.src` is part of an image object created in JavaScript. In creating the object, the URL (or just filename) of the source graphic is included as follows:

```
imageName.src= "graphicName.gif"
```

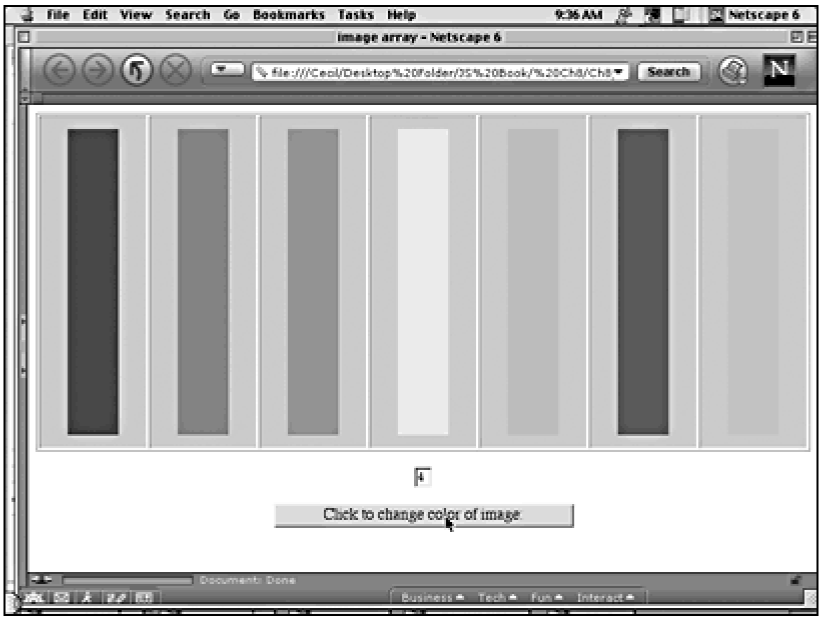
Because JavaScript can only dynamically change the `src` value in `document.images[n].src`, the new value must be defined as an `src`, not the image name or URL itself.

```
<html>
<head>
<title>image array</title>
<script language="JavaScript">
function changeIt() {
      var c1=new Image();
      var n=document.forms[0].hue.value;
      n = parseInt(n);
      c1.src="c3.gif"
      document.images[n].src=c1.src;
}
</script>
</head>
<body>
<table border cols=7 width="100%" height="75%" bgcolor="#cccccc" >
<tr>
<td align=center valign=center><img name ="c1" src="c1.gif" border=0
height=300
width=50></td>
<td align=center valign=center><img name ="c2" src="c2.gif" border=0
height=300
width=50></td>
<td align=center valign=center><img name ="c3" src="c3.gif" border=0
height=300
width=50></td>
<td align=center valign=center><img name ="c4" src="c4.gif" border=0
height=300
width=50></td>
<td align=center valign=center><img name ="c5" src="c5.gif" border=0
height=300
width=50></td>
<td align=center valign=center><img name ="c6" src="c6.gif" border=0
height=300
width=50></td>
<td align=center valign=center><img name ="c7" src="c7.gif" border=0
height=300
width=50></td>
</tr>
</table>
<center>
<form>
<input type=text name="hue" size=1><p>
```

```
<input type=button value="Click to change color of image:"
onClick="changeIt()";>
</form>
</center>
</body>
</html>
```

Figure 8.2 shows how the different color bars should appear on the screen and shows the window to enter the image array element value.

Figure 8.2. The HTML page contains the images array—JavaScript can address the array and change the `src` values.


File Edit View Search Go Bookmarks Tasks Help
9:36 AM
Netscape 6
image array - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/%20Ch8/Ch8
Search
Click to change color of image
Document: Done
Business Tech Fun Interact

Although other properties in the `Document` object can be addressed, such as the `link[]` and `anchor[]` arrays, the most important for the designer is the `images[]` property.

Preloading Images

When designing a page where one graphic replaces another, the image swapping should be immediate, and the viewer should not be required to wait while the new image loads. Fortunately, in JavaScript, preloading or placing images in the browser's cache is simple. First, a new image object is defined, and then the new object's source is defined as the following shows:

```
var niftyImage = new Image();
niftyImage.src = "coolPix.jpg";
```

That's all there is to it. The image is now cached in the browser—it's preloaded. Keeping in mind that the images object can be treated like an array, you can place the preloaded object in an HTML-defined image slot. For example, if you have an HTML line like the following:

```
<img src = "firstPix.jpg" name="firstUp">
```

you can replace it with the cached image using this line:

```
document.images[0].src = niftyImage.src;//images[0] is the first
image
```

or

```
document.firstUp.src= niftyImage.src;
```

There is no limit to the number of images that you can cache, but keep in mind that it will take the page longer to preload more than fewer images. Also, in caching your images, you can include the height and width as well. To generate a seamless replacement, the original and replacing objects should be the same. For instance, the following represent a good match between a JavaScript cached image and HTML images:

```
<img src = "firstArt" height= 87 width= 55 name="picasso">
var expArt = new Image(87,55)
```

The preloaded image has the same dimensions as the one loaded by HTML.

The DOM Connection

In an HTML page, the HTML structure contains elements and attributes. JavaScript treats the elements, such as images and forms, as objects, and it treats the elements' attributes as properties. For example, Table 8.4 shows the attributes of the `<IMG>` tag in HTML:

Table 8.4. HTML Attributes of the IMG Element

src	alt	longdesc
Align	height	width
Border	hspace	vspace
Usemap	ismap	

In JavaScript, the `<IMG>` element is treated as the `images[]` object, and all of the attributes of the `<IMG>` element are treated as properties of the `images[]` object. While JavaScript can read all of the image's properties, it can change only the `src` property.

With *all other* elements and their attributes in HTML that appear on the page (document), they are part of the DOM. JavaScript's relationship to the document's objects lies in the reference to HTML elements as objects and the attributes as properties of the referenced object. So, while the structure of the page lies in the HTML elements and attributes, the *behavior* of the page lies with JavaScript's capability to dynamically change certain elements' (objects') attributes (properties).

Summary

Objects in JavaScript are core, and understanding the document object model of JavaScript in relationship to HTML is essential to working effectively with the many objects that can be found on a web page. The idea behind object-oriented programming originally was to harness huge multiprogrammer projects because thousands of lines of code had to be coordinated. With the typical kind of projects that most designers will confront, the need for object-oriented programming is not to harness a giant project because most JavaScript is relegated to working with limited objectives in single pages. However, the extent to which you begin taking even baby steps in the direction of object-oriented programming, you will better understand what is occurring on your web page and how to make it do what you want.

A web page designer should be able to imagine the results of a dynamic web page and then, using JavaScript, make that web page come alive. By understanding OOP and even the concept of objects, properties, and methods in JavaScript, you have taken a major stride in that direction. Do not view objects and OOP as a burden, but rather as an opportunity to help you better realize what lies in your imagination. By the same token, do not treat OOP as an impediment to creativity. On several occasions, you just will not be able to figure out how to solve a problem using OOP, but the solution can be found using variables and statements that do not include objects linked to properties and methods. The OOP police won't come and get you! Take little steps in OOP, and eventually you will be able to harness its full potential.

Chapter 9. Frames and Addressing Frames in Windows

CONTENTS>>

The Window as a Complex Object

Chapter 7, "Objects and Object Hierarchies," covered most of the key properties and methods in the window object. As you saw in the discussion of the HTML hierarchy, at the very top is the window object. However, collected with the window object are other objects, including `frames`, `top`, `parent`, and `self`.

This chapter examines using and controlling frames within the window object. JavaScript not only can direct commands to other frames within a given window, but it also can pass data back and forth between frames. You can even dynamically create a page within a frame using JavaScript.

The Frame Elements and Attributes

The first order of business is to review the HTML elements and related attributes of a frame. Keeping in mind that, for JavaScript, an element is referenced as an object and each of the element's attributes is referenced as a property, the HTML structure is a blueprint for JavaScript behaviors. Table 9.1 shows the main frame elements and their attributes.

Table 9.1. Frame Elements and Their Attributes

Frame	Frameset	Iframe	Noframe
srcrows	src		
name	cols	name	
marginwidth	onload	marginwidth	
marginheight	onunload	marginheight	
scrolling		scrolling	
noresize		align	
frameborder		height	
	width		

Of all of the HTML frame elements, the key one is the frame object itself. Like images and forms, all window objects have a `frames[]` array. The array is a property of window and is referenced with the window's parent property. Thus, you may reference the third frame in a window as follows:

```
parent.frames[2]
```

If a frame is within another frame, that frame's parent is the frame it is within. If you want to address the top-level window and a frame within that window, you need to address the top object in the window object like this:

```
top.frames[1]
```

To address a subframe, you can reference the parent frame by its array element value and the subframe in the same manner. For example, if you have a frame within the fourth frame of the top level, you could address the target frame as follows:

```
frame[3].frame[4]
```

However, you can reference frames by their names. Note in Table 9.1 that one of the frame attributes is `name`. Just as with forms whose properties can be addressed by names, so can frames. To see how this works and some other addressing tricks, the following frameset and pages will be used to demonstrate how you might go about creating scripts that reference different frames using JavaScript.

Setting Up the Frameset

The frameset is important in naming your frames because those names become a reference for JavaScript. This line in the script:

```
<frame name="header" src="head.html" border=0>
```

creates an object that is referenced in JavaScript as follows:

```
parent.header
```

Also note the order of the frames. The first frame is named `header`, but it also can be referenced as this:

```
parent.frames[0]
```

Likewise, the other two frames (shown in the next frameNameSet.html listing) can be referenced by their array element number.

frameNameSet.html

```
<html>
<head>
<Title>Head Set</title>
</head>
<frameset rows="20%,*" frameborder=0 framespacing=0 border=0>
<frame name="header" src="head.html" border=0>
<frameset cols="20%,*" frameborder=0 framespacing=0 border=0>
<frame name="menu" src="menu.html" border=0>
<frame name="data" src="info.html" border=0>
</frameset>
</frameset>
</html>
```

The Header Frame

The first frame loads a page containing very simple yet important information about sending data to another frame. Because each frame has its own HTML page, the effect is passing data from one page to another. This JavaScript line addresses `parent.data`; the rest is document information:

```
parent.data.document.forms[0].elements[0].value="Greetings";
```

What is unique up to this point is the capability of the script to put data into another page. Data from one page in a frameset can be passed to or read from another page within the same set.

head.html

```
<html>
<head>
<style type="text/css">
body {font-family: verdana}
</style>
<Title>Header</title>
<script language="JavaScript">
function message() {
      parent.data.document.forms[0].elements[0].value="Greetings";
      }
</script>
</head>
<body bgColor=#ee9caa>
<h3> Header</h3>
<form>
```

```
<input type=text>
<input type=button value="Redecorate" onclick="message()" >
</form>
</body>
</html>
```

The Menu Page

The page in the menu frame uses a function to pass data to the other two pages in the frameset. Not only does it pass a greeting to the data frame, but it also changes the contents (values) of the button object in the header frame. Basically, it is no different from the page in the header frame, other than that it sends data to two different pages at once.

menu.html

```
<html>
<head>
<style type="text/css">
body {font-family: verdana}
</style>
<Title>Le Menu</title>
<script language="JavaScript">
function changeThem() {
      parent.data.document.forms[0].elements[2].value="Hi from Menu";
      parent.header.document.forms[0].elements[1].value="Hello";
}
</script>
</head>
<body bgColor=#ef001b>
<h3>Menu</h3>
<form>
<input type=button value="Send Two!" onClick="changeThem()">
</form>
</body>
</html>
```

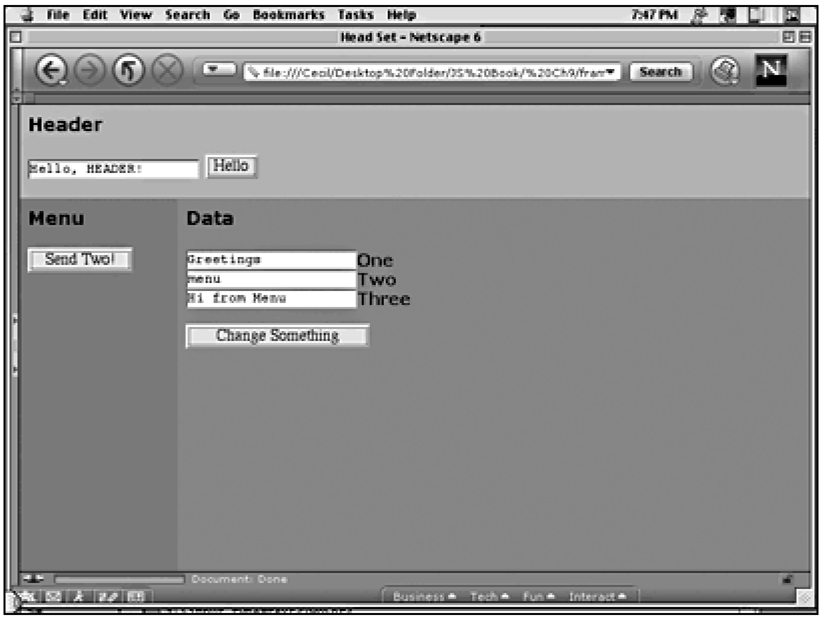
The Data Frame

The last frame has a couple of interesting actions. First, it gets data from the header frame by defining a variable as the name of `frame[0]`, which we know to be `header` with this statement:

```
hi=parent.frames[0].name;
```

Next, it sends a greeting in a string object that addresses the `header` frame by name. Figure 9.1 shows how this loud (all-caps) greeting looks. Finally, the script extracts the name from `frames[1]` and puts the name in one of its own form elements.

Figure 9.1. Data from properties in one frame can be passed to properties in other frames.


File Edit View Search Go Bookmarks Tasks Help
Head Set - Netscape 6
Search
Hello
Send Two!
Change Something
Document: Done
Business Tech Fun Interact

Header

Hello, HEADER!

Menu

Data

Greetings One
menu Two
Hi from Menu Three

info.html

```
<html>
<head>
<style type="text/css">
body {font-family: verdana}
</style>
<Title>Info page</title>
<script language="JavaScript">
function reply() {
      var hi=new String();
      hi=parent.frames[0].name;
      parent.header.document.forms[0].elements[0].value="Hello, " +
      hi.toUpperCase() + "!";
      var menName=parent.frames[1].name;
      document.forms[0].elements[1].value=menName;
      }
</script>
</head>
<body bgColor=#828793>
<h3>Data</h3>
<form> <h4>
<input type=text>One<br>
<input type=text>Two<br>
<input type=text>Three<p>
<input type=button value="Change Something" onClick="reply()" >
</form>
</body>
</html>
```

Scripts That Write Scripts

Up to this point, you have seen how to make changes by sending variables to different frames in a frameset. By passing variables between frames, you can basically establish communication between more than a single HTML page within a frameset. That is an important part of JavaScript, but what if you could dynamically alter an entire page? You could change everything from the text that appears on the screen to the entire layout of the page. This section shows how to do that.

Writing Elsewhere

Throughout the book, you have seen examples of using `document.write()` to place formatted text on a page. You can use the same method to write tags to create an entire page by addressing a document other than the one that you are using to write the JavaScript. Using the `window.open()` method, you can open a new window or target an existing one other than the one that you are using to write your script. The `open()` method has four arguments:

```
window.open(url,name,features,replace)
```

Up to this point, the only argument used is `url`. The `name` argument can be used to specify a name for a new window or a target window within a frameset. By using this format, you can target a frame to be the recipient of the output generated by using `document.write()`:

```
window.open("","frameName")
```

From the source page within a frameset, you create a function that targets a page in another frame using this general format:

```
function dynamic() {
      var varName = window.open("","frameName")
      varName.document.write('<tags and attributes>');
      varName.document.write(external variables);
      …
      varName.document.close();
}
```

Essentially, what this method accomplishes is to write a tag-based HTML page using `document.write()` to write the page. The function is launched from the page with the JavaScript. It opens another page, loads it with the tag-based script generated in the `document.write()` statements, and then closes the page. (Don't *ever* forget to close the page that you have opened!) Your last line in the function should *always* be `document.close()`.

A Basic Dynamic Page

To see how to use the dynamic page-creation capabilities of JavaScript, the following is a simple frameset and two pages that do two things:

- Change the background color of a page
- Print a message on the page

The fact that the changes are being made to a different frame and that the changes are dynamic makes this interesting, not the fairly pedestrian work of changing a background color or displaying a message using `document.write()`.

First, the frameset is unremarkable, but do note the names that have been given to the two frames. These names will be referenced in subsequent pages.

dynamicSet.html

```
<html>
<head>
<title> Dynamic Page Change </title>
</head>
<frameset cols="*,*" border=0 frameborder=0>
<frame name="message" src="message.html" border=0 frameborder=0>
<frame name="show" src="show.html" border=0 frameborder=0>
</frameset>
</html>
```

The core to this set of scripts is message.html. This page contains the JavaScript that will make the dynamic changes in the frame named `show` as defined in the frameset page. The main function first opens the window frame named `show`. (Actually, it would be more accurate to say that it *targets* the frame by name.) Next, it derives data from the current page and places the data into a variable named `message`. Then it starts using the `document.write` method to create a page beginning with the `<body>` tag. When it is finished writing the code for the page, the window is closed and the function terminates.

message.html

```
<html>
<head>
<title>Dynamic Change</title>
<script language="JavaScript">
function dynamic(){
      var dyn=window.open("","show")
      var message=parent.frames[0].document.forms[0].pass.value;
      dyn.document.write('<body bgcolor=#dddddd>');
      dyn.document.write('<center>');
      dyn.document.write('<p><p>');
      dyn.document.write('<h1>');
      dyn.document.write(message);
      dyn.document.write('</h1>');
      dyn.document.write('</center>');
      dyn.document.write('</body>');
      dyn.document.close();
      }
</script>
</head>
<body >
<p><p>
<h3>Enter your message in the text window:</h3>
<form>
<input name="pass" type="text"><p>
<input type=button value="Change the Page"onclick="dynamic()">
</form>
</body>
</html>
```

The final page, show.html, is really a dummy page for the purposes of this exercise. All it does is show that a text message and a distinct background color (hot pink!) currently appear on the page. Both the message and the background color get changed.

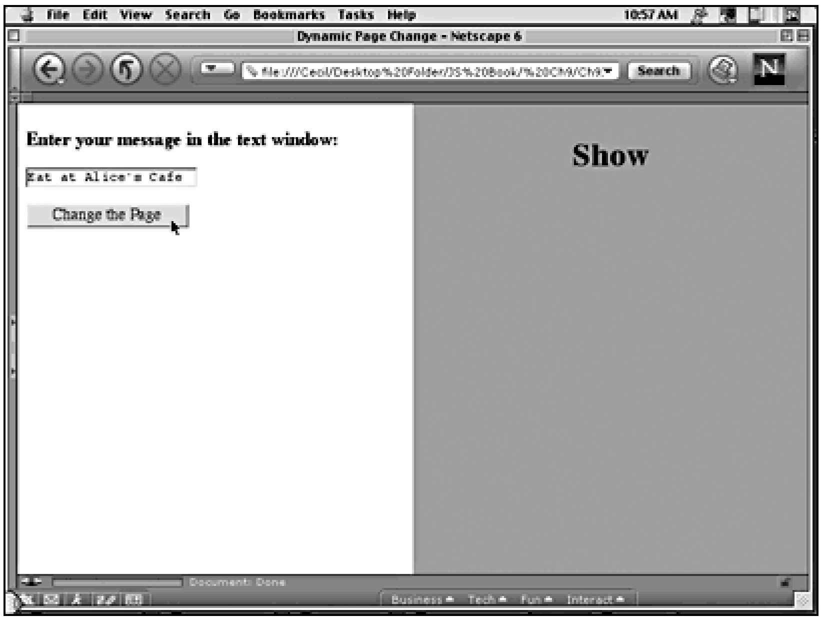
show.html

```
<html>
```

```
<body bgcolor="hotpink">
<center>
<p><p>
<h1>
Show
</h1>
</body>
</html>
```

Figure 9.2 shows the original frameset and pages as they appear when first loaded. Figure 9.3 shows what dynamic changes were made by the JavaScript in the page in the left frame.

Figure 9.2. The initial frameset shows two columns with a message indicating the name of the frame and page.


File Edit View Search Go Bookmarks Tasks Help
10:57 AM
Dynamic Page Change - Netscape 6
Search
Document: Done
Business Tech Fun Interact

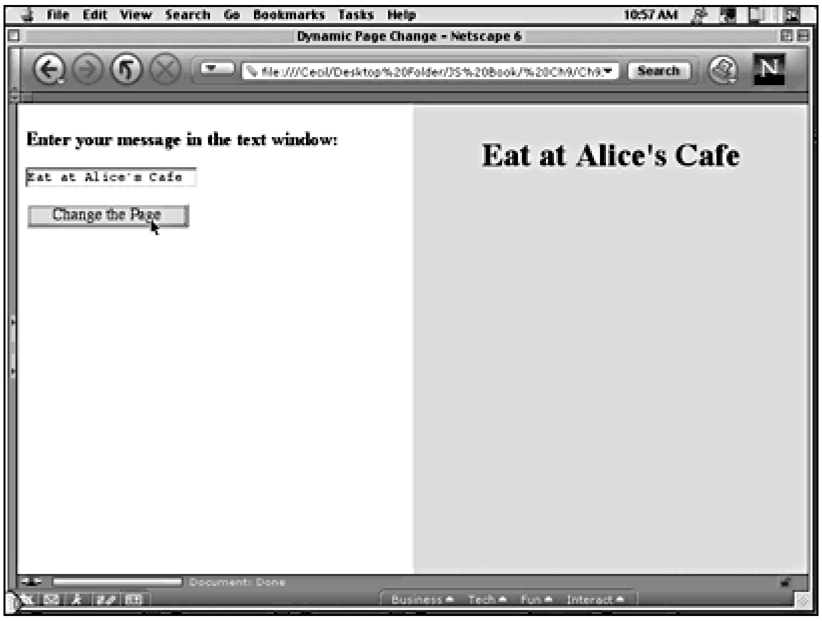
Enter your message in the text window:

Eat at Alice's Cafe

Change the Page

Show

Figure 9.3. The script from the page in the frame on the left dynamically changes the frame on the right.


File Edit View Search Go Bookmarks Tasks Help
10:57 AM
Dynamic Page Change - Netscape 6
Search
Document: Done
Business Tech Fun Interact

Enter your message in the text window:

Eat at Alice's Cafe

Change the Page

Eat at Alice's Cafe

More Dynamic Frames

The initial example of using dynamic frames shows only two changes made in the page, and it has little practical use. This next example is far more elaborate and provides a practical tool for comparing font color combinations. It uses a Cascading Style Sheet for the entire frameset, and then it uses `document.write` in combination with the `<FONT>` tag to change the colors in the fonts themselves. Another frame provides both the name and hexadecimal values for trying out different color combinations. An external CSS page provides all of the pages in the frameset with a consistent color and style.

color.css

```
body {
background-color: #decea1;
font-family: verdana
}
.header {
font-size:18pt;
font-weight: bold;
color: red
}
.explain {
font-weight: bold;
font-size: 11;
color: #be881d;
}
.label {
font-size: 14pt;
color: #7a8215
}
.bdtext {
font-size: 10;
color: #7a8215
}
```

The frameset has two columns and two rows in the second column. The first frame (`frame[0]`) uses 60% of the page for an entry display where the user can put in her color selections. The second frame (`frame[1]`), in the top row of the second column, shows several available names and their accompanying hexadecimal values. In the bottom-right corner is the frame named `colorfont` (`frame[2]`) that will be the target frame for the changes being devised in the first frame.

changeColorSet.html

```
<html>
<head>
<title> Frameset Create Page </title>
</head>
<frameset cols="60%,40%" border=0 frameborder=0>
<frame src="entry.html" border=0 frameborder=0>
<frameset rows="85%,15%" border=0 frameborder=0>
<frame src="colorname.html" border=0 frameborder=0>
<frame src="bottom.html" name="colorfont" border=0 frameborder=0>
</frameset>
</frameset>
</html>
```

The entry.html page has all of the code to write to the `colorfont` frame. Two different variables, `fcolor1` and `fcolor2`, store the HTML color name or hexadecimal value for the colors selected. These color values are passed to the font colors in the lines typified by the following:

```
doc.document.write('<font size= +3 color="'+ fcolor1 +'">');
```

The font color is specified by concatenating the variable name with the rest of the tag. The result on an HTML page for the previous line, with `blue` as the value of the variable `fcolor1`, would be the following tag:

```
<font size= +3 color="blue">
```

The trick is to get the double quotes around the color value or name. To get the double quotes around the color, the double quotes were placed within a set of single quotes.

entry.html

```
<html>
<head>
<link rel=stylesheet href="color.css">
<title>Dynamic Change</title>
<script language="JavaScript">
function showcolor(){
      var doc=open("","colorfont")
      var
fcolor1=parent.frames[0].document.forms[0].elements[0].value;
      var
fcolor2=parent.frames[0].document.forms[0].elements[1].value;
      doc.document.write('<body bgcolor=#dddddd>');
      doc.document.write('<center>');
      doc.document.write('<h3>');
      doc.document.write('<font size= +3 color="'+ fcolor1 +'">');
      doc.document.write("J");
      doc.document.write('</font>');
      doc.document.write('<font color="'+ fcolor2 + '">');
      doc.document.write("ava");
      doc.document.write('</font>');
      doc.document.write('<font size= +3 color="'+ fcolor1 +'">');
      doc.document.write("S");
      doc.document.write('</font>');
      doc.document.write('<font color="'+ fcolor2 +'">');
      doc.document.write("cript </h3>");
      doc.document.write('</font> <p>');
      doc.document.write('</center>');
      doc.document.write('</body>');
      doc.document.close();
      }
</script>
</head>
<body >
<center>
<form>
<p class="header">Interactive Font Color Changer</p>
<div class="label">Text Color 1&nbsp;&nbsp;<input
type="text"size="15"name="name">
```

```
<br>
Text Color 2&nbsp;&nbsp;<input type="text"size="15"name="name"></div>
<p>
<input type=button value="Color Fonts"onclick="showcolor()">
</form>
</center>
</body>
</html>
```

The next section of the script contains the names and the values of a sample set of HTML colors. The colors or the hexadecimal values can be copied and pasted from the color name frame to the text window in the browser to save time. The gray background was used to provide a neutral backdrop for the different color combinations.

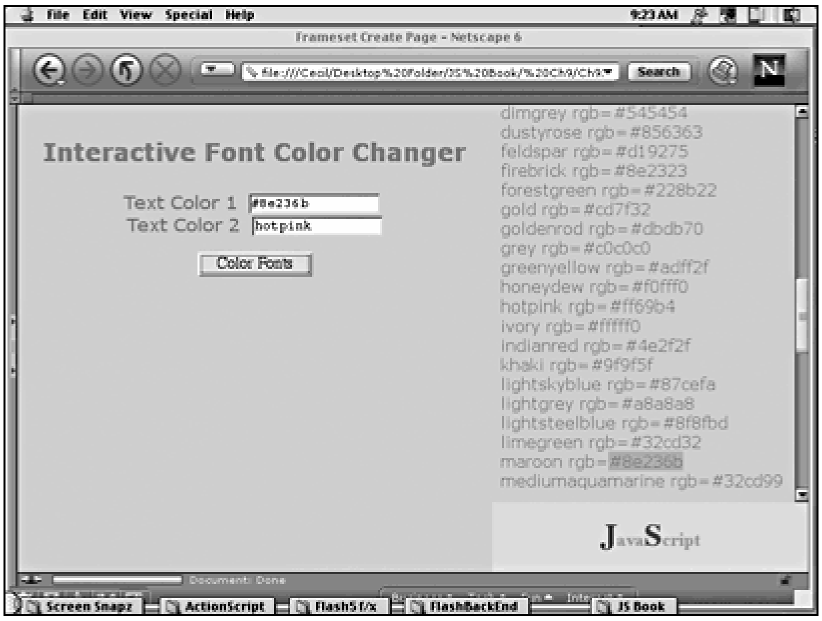
colorname.html

```
<html>
<head>
<link rel=stylesheet href="color.css">
<title> color names </title>
</head>
<body>
<p class="explain">The following are several HTML colors you might
want to use. Use
either the names or the hexadecimal values. If you use hexadecimal,
be sure to put the
pound sign (#) in front of the value (e.g., #2f4f4f.)</p>
<div class="bdtext">
white rgb=#ffffff
<br> red rgb=#ff0000
<br> green rgb=#00ff00
<br> blue rgb=#0000ff
<br> magenta rgb=#ff00ff
<br> cyan rgb=#00ffff
<br> yellow rgb=#ffff00
<br> black rgb=#000000
<br> aquamarine rgb=#70db93
<br> beige rgb=#f5f5dc
<br> bisque rgb=#ffe4c4
<br> blanchedalmond rgb = #ffebcd
<br> blueviolet rgb=#9f5f9f
<br> brass rgb=#b5a642
<br> brightgold rgb=#d9d919
<br> brown rgb=#a62a2a
<br> bronze rgb=#8c7853
<br> cadetblue rgb=#5f9f9f
<br> coral rgb=#ff7f50
<br> cornflowerblue rgb=#42426f
<br> darkgreen rgb=#2f4f2f
<br> darkorchid rgb=#9932cd
<br> darkpurple rgb=#871f78
<br> darkslateblue rgb=#7b68ee
<br> darkslategrey rgb=#2f4f4f
<br> darkseagreen rgb=#8fbc8f
<br> darktan rgb=#97694f
<br> darkolivegreen rgb=#556b2f
<br> darkorange rgb=#ff8c00
<br> blueviolet rgb=#9f5f9f
<br> darkturquoise rgb=#00ced1
<br> darkcyan rgb=#008b8b
```

```
<br> deepskyblue =#0bfff
<br> dimgrey rgb=#545454
<br> dustyrose rgb=#856363
<br> feldspar rgb=#d19275
<br> firebrick rgb=#8e2323
<br> forestgreen rgb=#228b22
<br> gold rgb=#cd7f32
<br> goldenrod rgb=#dbdb70
<br> grey rgb=#c0c0c0
<br> greenyellow rgb=#adff2f
<br> honeydew rgb=#f0fff0
<br> hotpink rgb=#ff69b4
<br> ivory rgb=#fffff0
<br> indianred rgb=#4e2f2f
<br> khaki rgb=#9f9f5f
<br> lightskyblue rgb=#87cefa
<br> lightgrey rgb=#a8a8a8
<br> lightsteelblue rgb=#8f8fbd
<br> limegreen rgb=#32cd32
<br> maroon rgb=#8e236b
<br> mediumaquamarine rgb=#32cd99
<br> mediumblue rgb=#3232cd
<br> mediumforestgreen rgb=#6b8e23
<br> mediumgoldenrod rgb=#eaeaae
<br> mediumorchid rgb=#9370db
<br> mediumseagreen rgb=#426f42
<br> mediumslateblue rgb=#7f00ff
<br> mediumspringgreen rgb=#7fff00
<br> mediumturquoise rgb=#70dbdb
<br> mediumvioletred rgb=#db7093
<br> midnightblue rgb=#191970
<br> navyblue rgb=#23238e
<br> orange rgb=#ff7f00
<br> orangered rgb=#ff2400
<br> orchid rgb=#db70db
<br> palegreen rgb=#8fbc8f
<br> pink rgb=#bc8f8f
<br> plum rgb=#eaadea
<br> salmon rgb=#6f4242
<br> scarlet rgb=#8c1717
<br> seagreen rgb=#2e8b57
<br> sienna rgb=#8e6b23
<br> silver rgb=#e6e8fa
<br> skyblue rgb=#3299cc
<br> slateblue rgb=#007fff
<br> springgreen rgb=#00ff7f
<br> steelblue rgb=#236b8e
<br> tan rgb=#db9370
<br> thistle rgb=#d8bfd8
<br> turquoise rgb=#adeaea
<br> violet rgb=#4f2f4f
<br> violetred rgb=#cc3299
<br> wheat rgb=#d8d8bf
<br> yellowgreen rgb=#99cc32
</div>
</body>
</html>
```

Finally, the dummy page simply awaits its fate to be changed. However, while it's waiting, it is dressed up in the same CSS as the other pages. Figure 9.4 shows what the viewer sees when all of these parts come together in the browser.

Figure 9.4. A designer can try out different color combinations without leaving the page.


File Edit View Special Help
9:23 AM
Frameset Create Page - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/%20Ch9/Ch9
Search
Document: Done
Screen Snapz
ActionScript
Flash5 f/x
FlashBackEnd
JS Book
JavaScript

Interactive Font Color Changer

Text Color 1 #8e236b
Text Color 2 hotpink

Color Fonts

```
dimgrey rgb=#545454
dustyrose rgb=#856363
feldspar rgb=#d19275
firebrick rgb=#8e2323
forestgreen rgb=#228b22
gold rgb=#cd7f32
goldenrod rgb=#dbdb70
grey rgb=#c0c0c0
greenyellow rgb=#adff2f
honeydew rgb=#f0fff0
hotpink rgb=#ff69b4
ivory rgb=#fffff0
indianred rgb=#4e2f2f
khaki rgb=#9f9f5f
lightskyblue rgb=#87cefa
lightgrey rgb=#a8a8a8
lightsteelblue rgb=#8f8fbd
limegreen rgb=#32cd32
maroon rgb=#8e236b
mediumaquamarine rgb=#32cd99
```

bottom.html

```
<html>
<head>
<link rel=stylesheet href="color.css">
</head>
<body>
</body>
</html>
```

Writing to Different Windows

In addition to writing to frames in a frameset, with JavaScript, you can also write to a different window. In Chapter 7, you saw how to open a separate window. In an almost identical manner that you can write to different frames using JavaScript, you can write to different windows.

The following script opens a window with a unique name, ralph, but without a URL. In other words, it's a new window, but not with a page to go with it. One new trick is introduced, but it's essentially the same kind of script and logic as you saw using the frameset. This line establishes the window, its name, and its dimensions:

```
var winR=open("","ralph", "scrollbars=0, width=200, height=100,
resizable=no")
```

The next line creates a shortcut when writing this script:

```
var s=winR.document;
```

Instead of having to type

```
winR.document.write();
```

you can now just type a much shorter version for each line because the rest of the object is contained in `s`:

```
s.write();
```

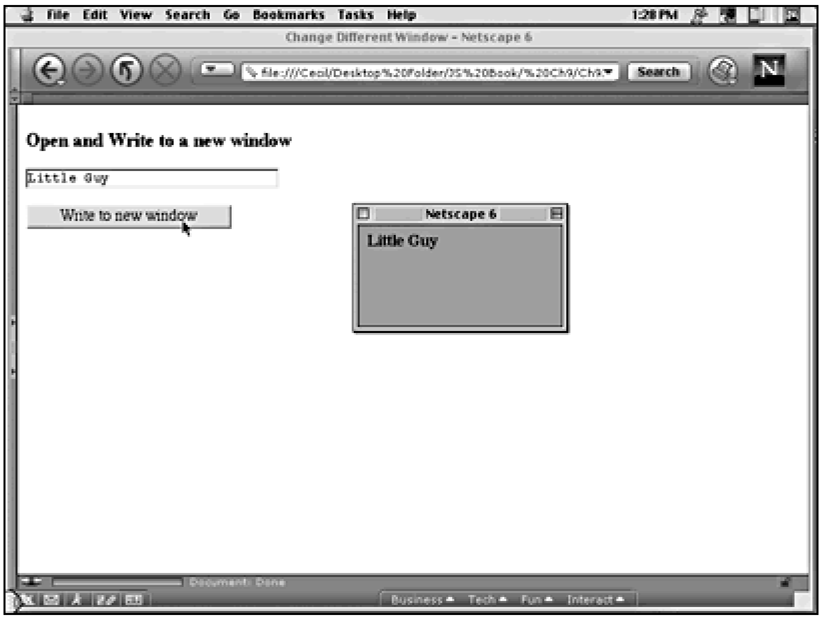
The only downside to this shortcut is that it is not quite as clear as writing out `winR.document.write()`. Besides, the best shortcut in JavaScript when you have a lot of repeated code that you cannot put into a loop is using cut and paste in your text editor.

winright.html

```
<html>
<head>
<title>Change Different Window</title>
<script language="JavaScript">
function fixWin(){
      var winR=open("","ralph", "scrollbars=0, width=200, height=100,
resizable=no")
      var s=winR.document;
      var message=document.forms[0].george.value;
      s.write('<body bgcolor="hotpink">');
      s.write('<h4>');
      s.write(message);
      s.write('</h4>');
      s.write('</body>');
      s.close();
      }
</script>
</head>
<body >
<form>
<h3>Open and Write to a new window</h3>
<input type="text" size=30 name="george">
<p>
<input type=button value="Write to new window" onclick="fixWin()">
</form>
</body>
</html>
```

Figure 9.5 shows that the message in the text window has been transferred to another window created with the script in the page. The default position of a new window is the upper-left corner of the window. The little window was dragged to the middle of the larger window but, with a bit more code, you could place it where you want on the screen.

Figure 9.5. JavaScript can write to other windows as well as other frames within a frameset.


File Edit View Search Go Bookmarks Tasks Help
Change Different Window - Netscape 6
Search
Write to new window
Netscape 6
Document: Done
Business Tech Fun Interact

Open and Write to a new window

Little Guy

Little Guy

Summary

The capability to pass data between different windows and frames means that data generated in one part of a web site can be used in another part. In applications in which data entered in one page is required in another page, JavaScript can be used as the agent of communication. While you will see in Part III, "JavaScript and Other Applications and Languages," that JavaScript in combination with various server-side scripts can do a great deal more in moving data over the web, JavaScript on its own can do a great deal as well.

One of the most interesting challenges for a web designer lies in dynamically creating pages, especially ones using data passed from the originating source. While it might seem a lot easier just to pass data between text or text area windows, you have no control over formatting the materials being passed. By dynamically changing pages, you can pass data and design the page while you're doing so.

Chapter 10. Event Handlers

CONTENTS>>

To have an *interactive* web page using JavaScript, you need two things: a user action and a reaction by the browser. The most common reaction in an HTML page is clicking the mouse, and you have seen several examples of the mouse firing off a function using the HTML `onClick` event handler in example scripts in this book. Some scripts have even included `onLoad` event handlers that fire a script automatically when a page is loaded. So by now, you have some idea about event handlers from your experience with basic functions such as rollovers.

This chapter takes a closer look at the events and event handlers in HTML pages and JavaScript. Here all of the events and the triggering conditions for an event to launch a function or JavaScript code are examined in detail. By the end of this chapter, not only will you have more options for firing functions, but you also will have a better understanding of how and when to use them. However, before getting into the events and event handlers, three unexamined window objects—`location`, `anchor`, and `history`— should be studied because all are often associated with events involving loading and unloading pages.

NOTE

One of the interesting characteristics of JavaScript and HTML is that, while they both use the same event-handler names, conventions have developed in HTML so that the event handles in HTML look different from those in JavaScript. HTML is not case-sensitive, and certain HTML conventions have led to spellings such as `onLoad` *and* `onMouseOver`*. However, in JavaScript, the event handlers are lowercased, such as* `onload` *and* `onmouseover`*. For the most part, the HTML event handlers in this chapter fire JavaScript functions, so the conventional upper/lowercase combinations are used. However, when an event handler in JavaScript is expressed as a property, you will see it done so only in lowercase.*

The *location, anchor,* and *history* Objects

Using links in HTML is second nature to web developers, and creating links with JavaScript is both easy and much more powerful than what can be done with the `<a href="myURL">` tag in HTML. The main linking objects in JavaScript are `location` and `history`. When the location object is assigned a URL, you get the same results as linking a page. It has the following format:

```
window.location="http//:www.yourURL.something";
```

(This states that the `window` object is optional, and it typically is not used with `location`. It is included simply to remind the reader that `location` is a property of `window`.)

The `location` object has eight properties and two methods, as shown in Table 10.1.

Table 10.1. Properties of the location Object

Properties	Methods
hash	reload()
host	replace()
hostname	
href	
pathname	
port	
protocol	
search	

To get a sense of the properties in the `location` object, the following scripts attempt to display all of the properties. The first script uses `location.URL` to load a page. The second script opens itself and displays information about its current status.

locationprop.html

```
<html>
<head>
<title>Location Properties </title>
<script language="JavaScript">
window.location="http://www.sandlight.com/JS/locaInfo.html"
</script>
</head>
<body>
</body>
</html>
```

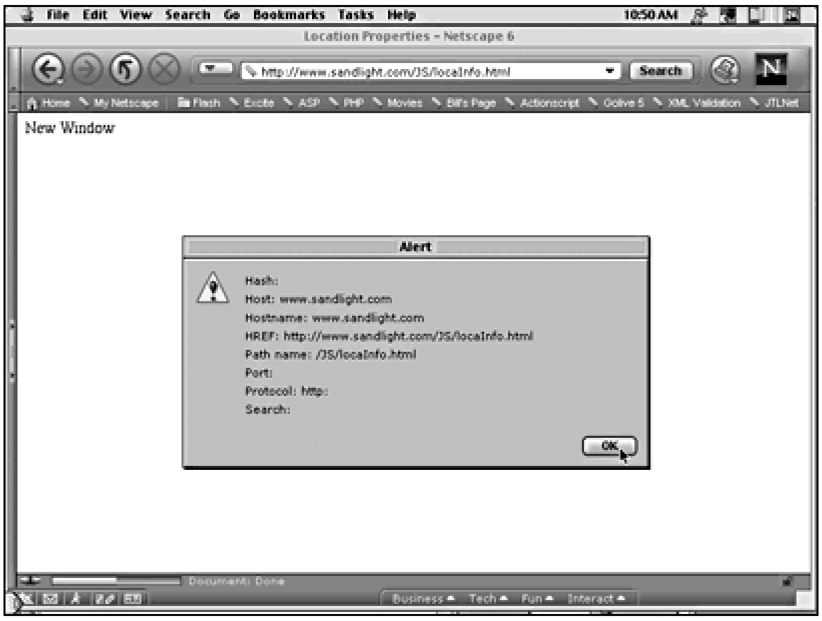
The second script is the page being called up for inspection. It will reveal information about its location in an alert box.

locaInfo.html

```
<html>
<head>
<title>Location Properties </title>
<script language="JavaScript">
function revealAll() {
      var cr="\n";
      var display = "Hash: " + location.hash + cr;
      display += "Host: " + location.host + cr;
      display += "Hostname: " + location.hostname + cr;
      display += "HREF: " + location.href + cr;
      display += "Path name: " + location.pathname + cr;
      display += "Port: " + location.port + cr;
      display += "Protocol: " + location.protocol + cr;
      display += "Search: " + location.search + cr;
      alert(display)
      }
</script>
</head>
<body onLoad="revealAll()">
New Window
</body>
</html>
```

As you can see from Figure 10.1, not all of the properties are displayed in the alert box.

Figure 10.1. The alert box displays all of the information about the location object available in the URL.


Location Properties – Netscape 6
http://www.sandlight.com/JS/locaInfo.html
Search
New Window
Alert
OK
Document: Done

```
Hash:
Host: www.sandlight.com
Hostname: www.sandlight.com
HREF: http://www.sandlight.com/JS/locaInfo.html
Path name: /JS/locaInfo.html
Port:
Protocol: http:
Search:
```

In looking at Figure 10.1, you can see that the page called was on a host named www.sandlight.com with the same hostname. The file's full HREF and pathname are shown along with the protocol. No port was cited, nor was a hash or search value returned. You will find such values in a URL like this:

```
http://www.sandlight.com:67/dogHouse/search.html?query=swissy&
matches=47#gsmd
```

The `port` value is `67`, the `hash` value is `#gsmd`, and the search value is `?query=swissy&matches=47`.

The two `location` methods, `locaton.reload()` and `location.replace()`, have very different functions. The `reload()` method acts just like the Reload button on a browser. However, the `replace()` method acts more like the `location` object itself. When a variable is defined in this format, it sends the page identified in the URL:

```
var sendMe=location.replace(URL);
```

However, the replacing page treats the previous page as though it never existed. If you press the Back button on the browser, it goes to the next-to-the-last previous page. The following script uses a menu to demonstrate the different ways to use the `location` property in a frameset context. A menu frame launches different pages delineated by a number and different cell background color to appear in an adjacent column.

The first script just sets up the frameset and provides target names for the two frames. The most important one to name is the second frame because no new pages will be replacing the page in the menu frame.

placeSet.html

```
<html>
<frameset cols ="10%,*">
```

```
<frame name="menu" src="placeMenu.html">
<frame name="travel" src="place.html">
</frameset>
</html>
```

The heart of the frameset is the menu script. In the script are examples of the different ways that `location` can be used to access a page in a frameset. Because only a single frameset is in the window, the script can use `parent` and `top` interchangeably. All of the JavaScript, except for a single function, is in the event-handler context. When you use several different single-statement scripts, it is just as easy to write the JavaScript in the event tag as it is to create a whole set of new functions.

placeMenu.html

```
<html>
<style type="text/css">
body {
      font-size:14pt;
      font-family:verdana;
      background-color:#ecdac0
      }
</style>
<head>
<script language="JavaScript">
    function goT() {
           parent.frames[1].location="place2.html";
    }
</script>
</head>
<body>
Menu:<p>
<form>
<input type=button value=" 1 " onClick = "parent.frames[1].location =
'place.html'"> <br>
<input type=button value=" 2 " onClick="goT()"><br>
<input type=button value=" 3 " onClick = "parent.travel.location =
'place3.html'"> <br>
<input type=button value=" 4 " onClick = "top.frames[1].location =
'place4.html'"> <br>
<input type=button value=" 5 " onClick = "top.travel.location
='place5.html'"> <br>
<input type=button value=" 6 " onClick =
"parent.frames[1].location.replace
('place6.html')"><br>
<input type=button value=" 7 " onClick ="parent.frames[1].location =
'place7.html'"><br>
</form>
</body>
</html>
```

This next script needs to be reproduced six times, for a total of seven pages. The first script is named place.html, the second is place2.html, the third is place3.html, and so forth until place7.html. Use the following cell-background colors, and replace the number to sequentially follow the filename numbers.

2 #537479

3 #eaaf54

4 #de5c0d

5 #cdb9a1

6 #704611

7 #782616

You will also want to change the text color value of the page display number in scripts 6 and 7 to `#ecdaco`. In that way, you can see the light numbers against the dark contrast of the cell background color.

NOTE

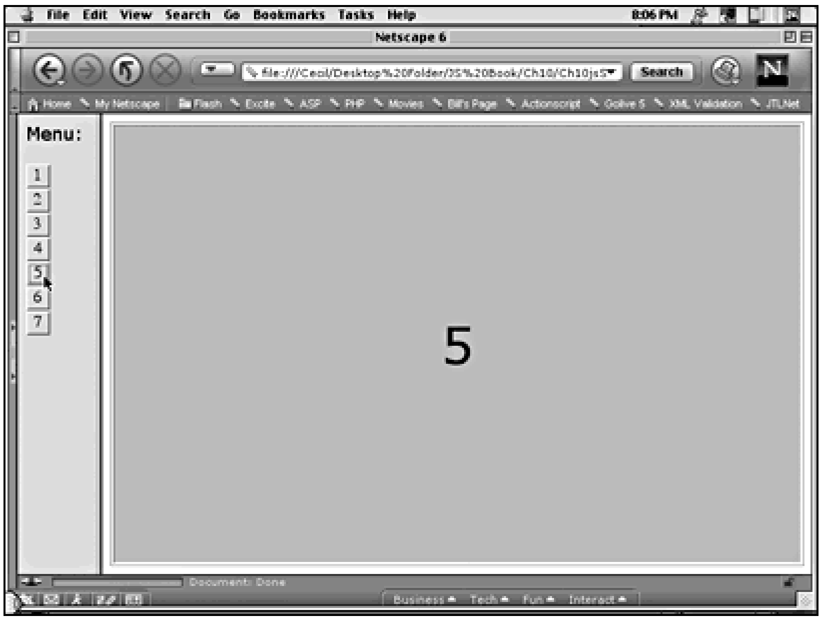
I used an old web page design trick with the pages in the right frame. Each page is dominated by a single-cell table. The numeric character in the middle of the page will maintain its relative position, no matter what size or proportion the frame is. By using the table and cell alignment attributes in HTML, it's easy to create a page that keeps its shape.

place.html

```
<html>
<head>
<style type="text/css">
.num {
      font-size:36pt;
      font-family:verdana;
      }
</style>
</head>
 <body>
<table width=100% height=100% align=center bgcolor="#f43615">
    <tr valign=center halign=middle>
       <td align=center height=”90%” width=90%><p class=”num”>1 </p>
       </td>
    </tr>
</table>
</body>
</html>
```

Figure 10.2 shows the frameset with the third page moved to the right frame by the script in Button 3.

Figure 10.2. All of the buttons fire a JavaScript function or statement to control the pages in the right frame.



Working with the *history* Object

The `history` object is a property of the `window` object. It has a single cross-browser property, `length`, and three cross-browser methods, `back()`, `forward()`, and `go(n)`. The length refers to the number of previous URLs that have been saved.

The most common use of the `history` object is to move back and forth in a site by referencing the previously visited sites. Depending on the order the pages were visited, you specify `back()` or `forward()`. The `history.go(n)` method uses positive values to go to a forward reference and negative numbers to go to a backward reference. Keeping in mind that the `history` object *can reference only previously visited pages,* the following two scripts demonstrate using the `history` functions. (Write both scripts first before trying to use them.)

The first script initially *has no effective history.* It has been nowhere, so it does not have a track record yet. You can click on the Go Forward button, and nothing will happen. However, after you go to the target page and return, the Go Forward button is active because it has a history to a forward page.

DblClick.html

```
<html>
<head>
<title>Forward </title>
<script language="JavaScript">
function doubleUp() {
      window.location="backPage.html";
      }
      function forwardTo() {
      var alpha=history.forward();
      }
</script>
</head>
<body bgcolor="lightgreen">
<p>
<H3> Remember this page!</h3>
```

```
First double-click the top button. That will take you to another
site.<br>
When you return to this page, click the bottom button.<P>
<form>
Double-click this button first:
<input type=button value="Two Clicks Here"
onDblClick="doubleUp()"><P>
When you return, click this button.<input type=button value="Go
Forward"
onClick="forwardTo()">
</form>
</body>
</html>
```

This second page has a history upon arrival because it is accessed from another page. It can "back up" to the page that it came from. Therefore, you have no need to build a history initially, and you can immediately fire the script that will return to the previous page.

backPage.html

```
<html>
<head>
<title>Reverse </title>
<script language="JavaScript">
function backUp() {
      var alpha=history.back();
      }
</script>
</head>
<body bgcolor="yellowgreen" onUnLoad="alert('You can return here by
pressing the Go
Forward button on the page.')" >
<p>
<H3> Thanks for the Visit</h3>
Place your mouse over the button and return from whence you came.<P>
<form>
<input type=button value="Mouse over here" onMouseOver="backUp()"><P>
</form>
</body>
</html>
```

anchor Object

The `anchors[]` array is somewhat of a disappointment at this time. The two major browsers are incompatible with anchors. Within the `location` object is a `hash` property that can be used to target an anchor with the following format:

```
location.hash="anchorName"
```

Within a page, `location.hash` can provide to be quite useful; however, all that the `anchors[]` array returns in a cross-browser environment is the `length` property with this format:

```
document.anchors.length
```

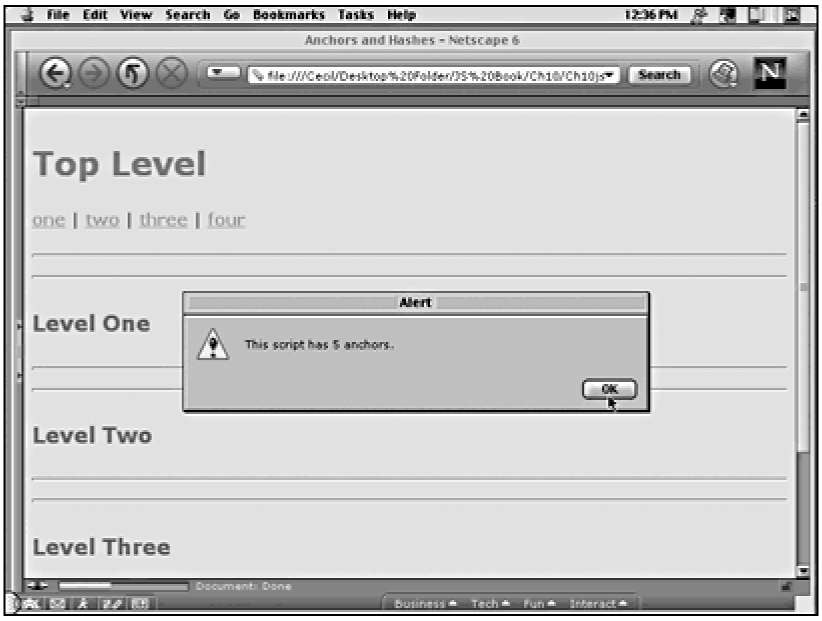
However, if you run the `anchors[]` array through a loop with `anchors.length`, you will not get the anchor names returned. The following script should give you a good idea of what you can do with anchors in a cross-browser environment. (Several different mouse events were used in preparation for the following section on event handlers.)

anchorsArray.html

```
<html>
<head>
<title>Anchors and Hashes </title>
<style>
body {
      background-color:#d9e6cc;
      font-family: verdana;
      }
h2 {
      font-size: 24pt;
      color: #ff0040;
      }
h3 {
      font-size: 16pt;
      color: #008080;
      }
a {color:#e68026; font-size:10; font-weight:bold;}
</style>
<script language="JavaScript">
function countAnchors() {
      alert("This script has " + document.anchors.length + "
anchors.");
      }
function goAnchors(anchName) {
      location.hash=anchName;
      }
</script>
</head>
<body onLoad="countAnchors();">
<p>
<h2>Top Level <a name="top"></h2>
<a href="#" onMouseDown="goAnchors('one')">one</a> | <a href="#"
onMouseOut =
"goAnchors('two')"> two </a> | <a href="#" onMouseMove
="goAnchors('three')"> three </a>
|
<a href="#" onMouseOver="goAnchors('four')">four</a>
<p>
<p><hr><hr>
<p>
<h3>Level One<a name="one"></a></h3>
<p><hr><hr>
<p>
<h3> Level Two<a name="two"></a> </h3>
<p><hr><hr>
<p>
<h3> Level Three<a name="three"></a> </h3>
<p><hr><hr>
<p><a href="#" onMouseUp="goAnchors('top')">Top</a>
<h3> Level Four<a name="four"></a>
</h3>
</body>
</html>
```

Figure 10.3 shows that the `anchors[]` array does, in fact, have elements, but getting to the elements in JavaScript has not been consistently resolved.

Figure 10.3. The `anchors[]` array has limited use, but the `location.hash` object/property can target anchors.


File Edit View Search Go Bookmarks Tasks Help
12:36 PM
Anchors and Hashes - Netscape 6
file:///Ceol/Desktop%20Folder/JS%20Book/Ch10/Ch10js
Search
Alert
OK
Document: Done
Business Tech Fun Interact

Top Level

one | two | three | four

Level One

This script has 5 anchors.

Level Two

Level Three

Events and Event Handlers in HTML and JavaScript

Throughout the book, and especially in this chapter, you have seen examples of different event handlers, and by now you should have an idea of what they do. To complete your understanding of what events are, how they are triggered, and what you can do with them, this chapter breaks them down into four different categories:

- Mouse events
- Key events
- Form events
- Page/window/image events

Some events and event handlers are different in the two major browsers. As with the other topics in this book, the focus will be on those features of JavaScript that have cross-browser compatibility. Likewise, some of the events deal with certain elements in HTML, to the extent that much of the discussion of those events is reserved for the chapters that cover the specific topics where the event handler has the most relevance. Most notable are the form events that are covered extensively in the next chapter (Chapter 11, "Making Forms Perform"). Event-like properties, such as `form.checked`, not only will be examined in the next chapter to look at the particular object with which they are most pertinent, but they also will be treated as *properties* and not *events.* In many ways, the event handlers that can be treated as properties in JavaScript represent the future of client-side JavaScript, and as the language (not to mention the browser manufacturers) matures, we hope to see cross-browser implementation of event handlers that are methods (a type of property), not as *external* HTML firing devices.

Mouse Events

Seven mouse events can be used to trigger a JavaScript program. They include the following:

- `onClick`
- `onDblClick` (NN6+)
- `onMouseDown`
- `onMouseUp`
- `onMouseMove` (NN6+)
- `onMouseOut`
- `onMouseOver`

Most of the mouse events listed are familiar to you by now because so many have been used in examples. The following script provides an example of each of the mouse events, with the appropriate responses for feedback:

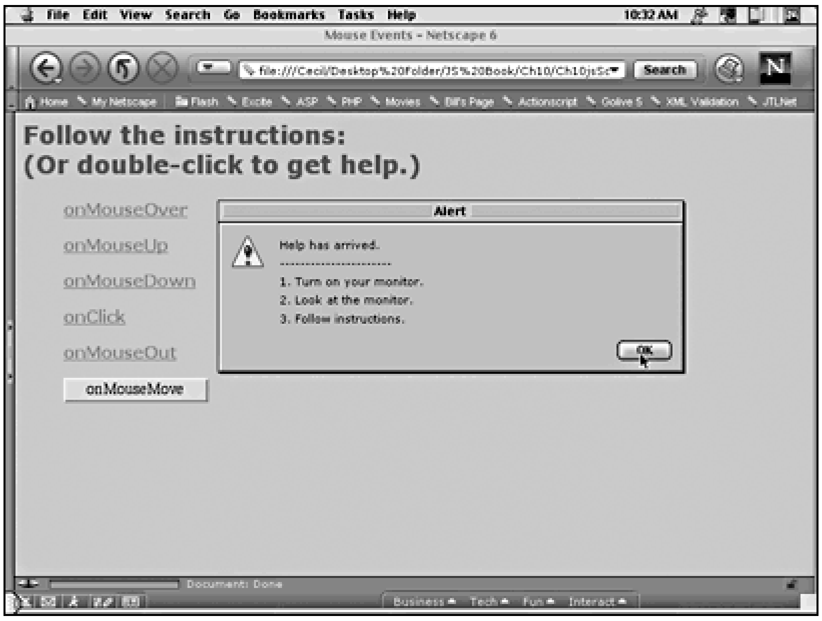
```
<html>
<head>
<title>Mouse Events </title>
<style type="text/css">
body{
     font-family:verdana;
     background-color:#ffbf00;
     }
h2 {
    color:#b51300;
    font-size:16pts;
    font-weight:bold;
}
a {color:a573b6;font-weight:bold}
</style>
<script language="JavaScript">
function helpMe() {
      alert('Help has arrived. \n----------------------\n 1. Turn on
your monitor.
      \n 2. Look at the monitor.\n 3. Follow instructions.');
      }
</script>
</head>
<body onDblClick="helpMe()";>
<H2> Follow the instructions: <br> (Or double-click to get
help.)</h2>
<ul>
<a href="#" onMouseOver="alert('As you move over the spot.');">
onMouseOver </a><p>
<a href="#" onMouseUp="alert('You have to press Down to get Up');">
onMouseUp </a><p>
<a href="#" onMouseDown="alert('You can keep it down.');">
onMouseDown </a><p>
<a href="#" onClick="alert('Any click will do');"> onClick </a><p>
<a href="#" onMouseOut="alert('First move over and then off');">
onMouseOut </a><p>
<form>
<input type=button value ="onMouseMove" onMouseMove="alert('Press
enter or return--
don\'t move the mouse!')"><p>
```

```
</ul>
</form>
</body>
</html>
```

The script uses the double-click (`onDblClick`) as part of the `<body>` tag. You're accustomed to seeing the `onLoad` event handler in the `<body>` tag, but you can place any of your mouse or key event handlers there as well. By using the double-click event handler, there is less chance of the event cropping up in one of the single-click hot spots on the page.

Figure 10.4 shows all the different mouse hot spots, a button, and what happens when the user double-clicks the mouse anywhere on the page.

Figure 10.4. The mouse event handlers can be placed in most HTML tags that recognize an event, including the `<body>`, `forms`, and `<a>` tags.



Follow the instructions:
(Or double-click to get help.)

Help has arrived.
----
1. Turn on your monitor.
2. Look at the monitor.
3. Follow instructions.

Key Events

JavaScript recognizes three key events:

- `onKeyDown`
- `onKeyUp`
- `onKeyPress`

Unfortunately, Netscape Navigator and Internet Explorer have different ideas on how to implement certain key features (no pun intended) for these events. As a result, the designer has a limited choice of options for a bulletproof cross-browser implementation of the key event. The general format of any of the key events resides in firing them from within different HTML tags:

```
<tag onKeyEvent="functionToFire()">
```

You can place the key events in any of the tags that accept mouse events, but capturing *which key has been pressed* is inconsistent between the browsers. As a result, the event handlers have limited utility. The following script shows several different ways that the key event handlers can be employed. Because one of the key events is placed in the `<body>` tag, it pops up whenever any of the other key events are triggered.

```
<html>
<head>
<style type="text/css">
body{
     font-family:verdana;
     background-color:#e3f37f;
     }
h2 {
     color:#4d57ab;
     font-size:16pts;
     font-weight:bold;
}
a {color:fe0006;font-weight:bold}
</style>
<script language="JavaScript">
function fillIn() {
      document.forms[0].elements[1].value="JavaScript is here!"
}
</script>
<title>Keys Up and Down </title>
</head>
<body bgColor= "cornflowerblue" onKeyPress="alert('This was set up in
the body
element.');">
<center>
<H2> Click on the link text, button, or text window.<br>
Then press any key.</h2>
<a href="#" onKeyUp="alert('The <a href> did it!');"> Link Text
</a><p>
<form>
<input type=button value ="Surprise me" onKeyPress="alert('You have
to press the
Surprise Me button first!')"><p>
<input type=text size=20 onKeyDown="fillIn()">
</center>
</form>
</body>
</html>
```

When you press the `Link Text` hot spot, the event trigged by the `onKeyPress` event handler in the `<body>` tag appears and seems stuck. However, if you press the Enter/Return key on your computer, the key-launched function from the `Link Text` hot spot appears and keeps coming up every time you press a key. Now you have to click on the OK button in the alert box.

Form Events

The next chapter will explore form events, along with other form characteristics, in greater detail. In this section, however, you will see how to use the basic event handlers associated with forms. JavaScript recognizes five event handlers HTML associates with forms:

- `onBlur`
- `onChange`
- `onFocus`
- `onReset`
- `onSubmit`

The references to "blur," "change," and "focus" all pertain to *text elements in HTML* only. When using the `<input type=text>` tags in HTML, the resulting text windows can be clicked and text can be entered. At the time that text is entered in a window, the window is considered to be *in focus.* The focus event means that a particular text window has been clicked and is ready for text input. The general format for setting up an `onFocus` event handler in HTML is the following:

```
<input type=text onFocus="functionGo()">
```

As soon as the user clicks in the text box, the function fires. The following little script shows how to use `onFocus`:

```
<html>
<head>
<title>onFocus</title>
<script language="JavaScript">
function autoWrite() {
      document.forms[0].elements[0].value="Greetings earthling!";
      }
</script>
</head>
<h3>Click on the Text Window:</h3>
<body bgColor="rosybrown">
<form>
      <input type=text onFocus="autoWrite();">
</form>
</body>
</html>
```

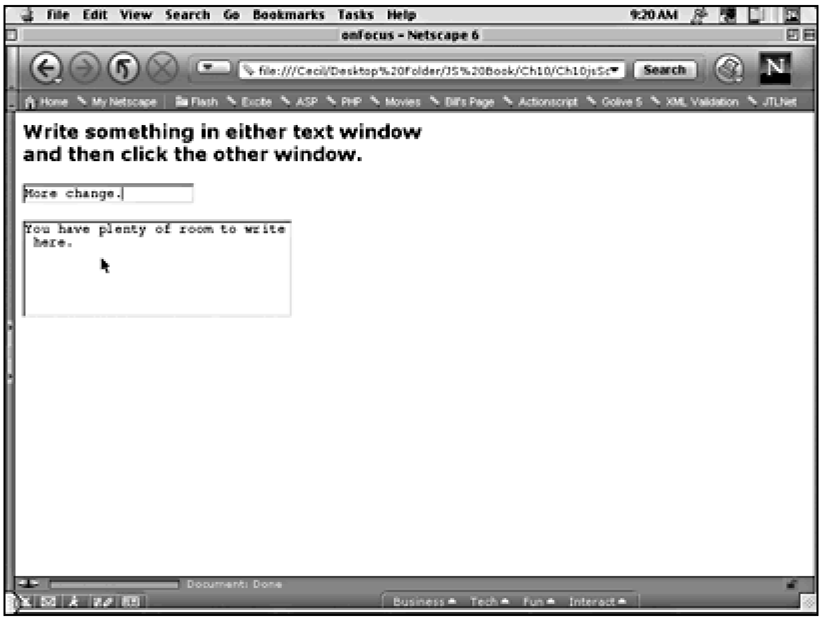
The other two event handlers that look for events in text window, `onChange` and `onBlur`, first look for the event of selecting a text window and then selecting something else. The blur event occurs when a window has been selected and then deselected, while the change event looks to see whether a window has been selected *or* deselected. The following script shows how both are used, in both a text element and a textarea element:

```
<html>
<head>
<style type="text/css">
body { background-color:peru; font-family:verdana; color:white}
</style>
<title>onFocus</title>
<script language="JavaScript">
var auto = new Object();
var bigOne= new Object();
```

```
function showChange() {
      auto=document.forms[0].ralph;
      auto.value="More change."
      }
function areaWrite() {
bigOne=document.forms[0].alice;
bigOne.value="You have plenty of room to write here."
}
</script>
</head>
<h3>Write something in either text window<br>
and then click the other window.</h3>
<body>
<form>
      <input type=text name="ralph" onChange="showChange();"><p>
      <textarea cols=30 rows=5 name="alice"
onBlur="areaWrite();"></textarea>
</form>
</body>
</html>
```

No matter what content is placed in either form element, they always end up with the same message because, as soon as the user moves out of the window, the functions fire. Figure 10.5 shows what the windows will always appear to be after a text window is clicked.

Figure 10.5. Both blurring and changing in text and textarea windows can launch JavaScript functions.



Write something in either text window
and then click the other window.

More change.

You have plenty of room to write here.

The final two event handlers associated with forms are unique in that they are a form *type* as well as an event handler. To establish a Reset and Submit button in a form container, use this format:

```
<input type=submit>
<input type=reset>
```

What’s more, the Reset button will clear all of the text and textarea windows in the form with nothing more than the tag `<input type=reset>`. No `onReset()` event handler is required, and no JavaScript is needed, either. However, using the `reset()` method in JavaScript allows you to reset the forms with the method attached. Also, the onreset event handler can be used as a property in JavaScript in a cross-browser environment. This next example shows how the `reset()` method can be treated as a property of a form. Because it is a property of a form, you will see that it clears *only* the form elements of which the method is connected. A second dummy form is provided so that you can see that, while one form is cleared, the other is not. Note also that the `reset()` method is not part of a form element (property), but is connected directly to the form object.

```
<html>
<head>
<style type="text/css">
body { background-color:darkorange; font-family:verdana; color:white}
</style>
<title>Reset Property</title>
<script language="JavaScript">
function autoFire() {
      document.hope.reset();
      }
</script>
</head>
<h4>Type in something in the both text windows and then<br>
pass the mouse over the "JavaScript Method" button.</h4>
<body >
<form name="hope">
      <input type=text name="faith"><p>
      <input type=button name="glory" value="JavaScript Method"
onMouseOver="autoFire();">
</form> <p>
<form name="dummy">
      <input type=text name="dummer">
</form>
</body>
</html>
```

In addition to the `reset( )` method, JavaScript has a `submit( )` method that coincides with `onSubmit( )` in HTML. However, to really understand the `submit( )` method or the `onSubmit( )` event handler, you will need to understand more about submitting forms. A discussion of the `submit( )` method is left to Chapter 11, where forms are explored in depth.

Page/Window/Image Events

As a category, this last set of events occurs as a side effect to what a user might have initiated. The following five event handlers make up this category:

- `onAbort`
- `onError`
- `onLoad`
- `onResize`
- `onUnload`

These event handlers work with changes that occur when an event occurs that is not directly controlled by the user. A click by the mouse is fully controlled by the user, but a closing page, while selected by the user because she decided to go to another page, occurs *incidental* to the event. (In this case, the event would be going to another page.) In several examples, the onLoad event handler has been used to illustrate different features of JavaScript and a few for onUnload, but the other event handlers in this category have not been discussed at all.

The onLoad and onUnload event handlers are associated with the page itself, so the <body> tag is where they are expected to be found—and where they usually are found. The event occurs *after* the page has been loaded, so references to forms that might be in the page that are parsed *after* the <body> tag can still be included in any function launched by the onLoad event handler. A lesser known use of the onLoad and onUnload event handlers is with the <image> tag. As soon as an image has completed loading, the event can be used to launch a function. Both tags use the following format:

```
<image src="someGraphic.jpg" onLoad="showLoaded( );">
<body onLoad="showLoaded( );">
```

The following example shows how the onLoad event handler can fill in a form and announce the loading of an image. (An intentionally large graphic file helps in understanding what happens in an onLoad situation.)

```
<html>
<head>
<title>Using Load with pages and images</title>
<script language="JavaScript">
         function whew( ) {
         alert("Finally got tubby loaded!");
         }
function fillEmUp( ) {
         var alpha="All done!";
         document.fuzzy.wuzzy.value=alpha;
         }
</script>
</head>
<body bgColor="floralwhite" onLoad="fillEmUp( );">
<img src="jumbo.jpg" onLoad="whew( )">
<form name="fuzzy">
<input type=text name="wuzzy">
</form>
</body>
</html>
```

Both of the onLoad events fire different functions, so you can see that each is working independently with the same type of event handler. The onUnload handler works the same way, except that it occurs on closing.

When something goes wrong, you might want to know about it, especially when debugging the program. Sometimes users will elect to abort a page themselves, most notably when the page is slow in loading. The onError and onAbort event handlers can help in both instances. Both event handlers are used in combination with the <image> tag. The first script shows where to place an onError event handler. (Be sure you do *not* have a file named importantPic.jpg in the folder where you save this next file.)

```
<html>
<head>
<title>Error</title>
<script language="JavaScript">
function whoops() {
         alert("Ok, Elrod, where\'d you put the graphic file?")
         }
</script>
</head>
<body bgcolor="lightskyblue" >
<p>
<h3>This is an important picture you will want to see!</h3>
<img src="importantPic.jpg" onError="whoops( );">
</body>
</html>
```

The `onAbort` event handler works in a similar manner to `onError`, except that, rather than an error causing the show to come to a halt, the viewer has elected to click the Stop button on the browser to stop the rest of the page from loading— usually because of a long load. In those cases, you might want to put an `onAbort` handler in the `<image>` tag with a message to the effect, that even though the graphic is large, it is certainly worth waiting for!

`onResize` is the last event handler examined in this chapter. The event is one that affects the window size, but the event handler can be placed in either the `<body>` or the `<frame>` tags. In web page design, information returned from `onResize` can be critical if linked to some other changes that you want to control in the design. However, the event handler itself is fairly straightforward. Run the following script, and change the size of the window to trigger the event:

```
<html>
<head>
<title>Size change</title>
<script language="JavaScript">
function sizeMeUp( ) {
      alert("Size change!")
      }
</script>
</head>
<body bgcolor="maroon" onResize="sizeMeUp( );">
<p>
<h3>Nothing to see here folks! Try resizing the window.</h3>
</body>
</html>
```

Summary

Event handlers are the interactive triggers in HTML and JavaScript. Whether the script traces a location or history property or launches a function, the World Wide Web would be far less interactive and interesting without the event handlers. They make things happen and give the designer tools to use events in planning and executing a web site design.

The bulk of the event handlers do not now have good cross-browser compatibility when used as properties of JavaScript objects. In future versions of JavaScript, such as JavaScript 2.0, developers and designers fervently hope that all browsers will follow the new standards consistently and will implement better event-handling properties that will launch from a method rather than HTML. However, rather than cursing the darkness of cross-browser incompatibility, using the nonconflicting event handlers in a coordinated development with an HTML page provides a wide variety of events from the mouse, keyboard, forms, and pages. By using the rich mix of events, designers can set up a web environment in which the user can experience a wide range of experiences that she herself causes to happen.

Chapter 11. Making Forms Perform

CONTENTS>>

Forms used in conjunction with JavaScript can provide a wide source of engaging and interactive pages for the designer. Besides being used as crucibles for information such as name, address, and email, forms are used to respond to different activities by the user. Everything from games to quizzes to online calculators can be created using HTML forms and JavaScript together. So, if you thought forms were boring, take a look and see what you can create using them with JavaScript.

Two critical tasks await JavaScript. First, JavaScript can take the data within the forms and pass it through variables from one place to another on an HTML page or even to different HTML pages. By passing form data in variables, JavaScript can be more precisely interactive with HTML than a generic set of data. Second, JavaScript serves as a form-verification medium. All data headed for a database or back-end middleware like ASP and PHP can first be checked for accuracy and completeness by JavaScript. For example, JavaScript can make sure that any email addresses that are entered into a form have the @ sign and, if the @ sign is missing, alert the viewer to that fact. Beginning in Chapter 14, "Using PHP with JavaScript," where the book begins examining server-side scripting with JavaScript as an intermediary, you will see how the form data can be checked first by JavaScript before the variables in the forms are sent to the back end.

The Many Types of Forms Elements in HTML

The `forms` element in HTML is more like a mega-element with many other elements within. Each of the HTML forms elements has a number of properties, which in HTML are known as "attributes."

- forms element (mega-element)
- forms elements (e.g., text boxes, buttons, etc.)
- form attributes (e.g., name, size, width, etc.)

In looking at an HTML page with a form, with the exception of the `<textarea>` container, the only container within the `forms` object is the form itself. That is, the `forms` element has an opening and closing tag, but none of the elements within the container has closing tags:

```
<form>
      <form element 1>
      <form element 2>
      <form element 3>
</form>
```

As far as JavaScript is concerned, the `forms` object is one big array. Each new form container is an element of the document array, and each element within each `forms` object is an element of that form's array. Later in this chapter, form arrays are examined in detail, but for now, it is important to understand that an array relationship exists between JavaScript and HTML forms.

The following list summarizes all of the form objects in HTML tags, arranged by type:

HTML element

Text input

- `<input type=text>`
- `<textarea> </textarea>`
- `<input type=hidden>`
- `<input type=file>`
- `<input type=password>`

Buttons

- `<input type=button>`
- `<input type=reset>`
- `<input type=submit>`
- `<input type=radio>`
- `<input type=checkbox>`
- `<input type=image>`

Menu

- `<select>`
- `<select multiple>`
- `<option>`

JavaScript's relationship to the HTML tags has been illustrated in numerous examples in previous chapters, but it bears repeating. As a general format, the relationship between the objects and properties in JavaScript is the same as between the form order and name attributes in HTML. The following HTML tags:

```
<form name="alpha">
      <input type =text name="beta">
```

```
</form>
```

are addressed like this in JavaScript:

```
document.alpha.beta;
```

Attributes of the secondary elements (forms is the primary element) are treated as properties of the secondary elements. For example, if you wanted to know how long a string in a text box is, you would address it as follows:

```
document.alpha.beta.value.length;
```

You might wonder why the `value` *and* the `length` properties had to be included instead of just `length`. The property itself has no length unless you're looking for the length of an array. However, the value assigned to the property *does have* a length, and that is what you want. If you entered this, you could find out the length of the array, not any of the values in the elements of the form:

```
document.alpha.length;
```

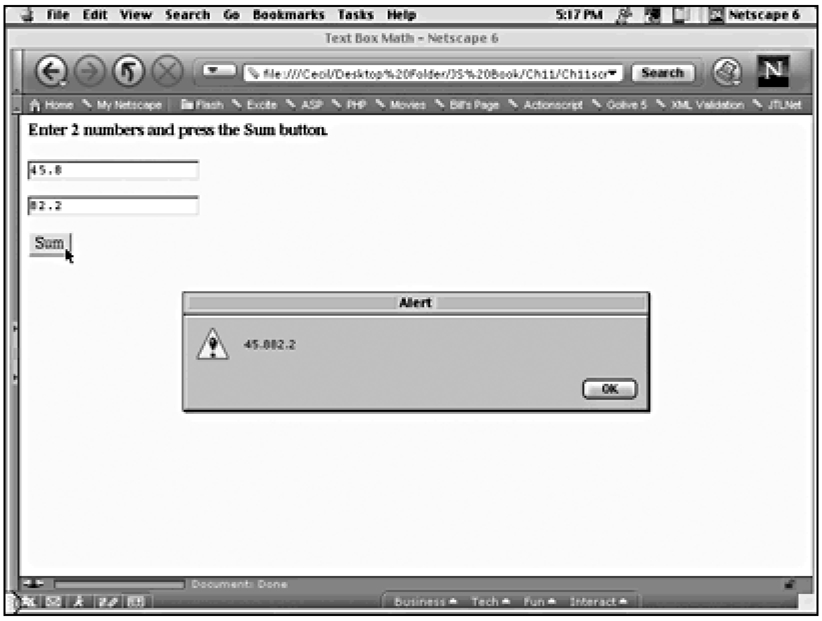
All Text Entries Are Strings

Another feature of form data to remember is that it treats all entries as strings. For example, in setting up an online boutique, you might want to perform math on data entered by the user. The following little script attempts to add two numbers:

```
<html>
<head>
<title> Text Box Math </title>
<script language="JavaScript">
function addEmUp( ){
      var sum=document.calc.aOne.value + document.calc.aTwo.value;
      alert(sum);
      }
</script>
</head>
<body bgcolor="#ffffaa">
<h4>Enter 2 numbers and press the Sum button.</h4>
<form name="calc">
<input name="aOne" type=text ><P>
<input name="aTwo" type=text ><P>
<input type=button value="Sum" onClick="addEmUp( )">
</form>
</body>
</html>
```

When you run the script, you will see, as shown in Figure 11.1, that instead of getting a sum, you get a string of two numbers. The script concatenated two strings instead of adding two numbers.

Figure 11.1. Instead of adding numeric values from text fields, JavaScript concate nates them.


Enter 2 numbers and press the Sum button.

45.8

82.2

45.882.2

While numbers are unsigned (they are neither integers nor floating point) in JavaScript, strings that are transformed into numbers are converted into either integers or floating-point numbers using one of the following functions:

```
parseInt( )
parseFloat( )
```

For example, if your string is `34.763`, use `parseFloat ( )` to preserve the decimal points, or use `parseInt ( )` to return the integer, rounded down. By changing this line:

```
var sum = document.calc.aOne.value + document.calc.aTwo.value;
```

to

```
var sum = parseFloat(document.calc.aOne.value) + parseFloat
(document.calc.aTwo.
value);
```

your output changes to reflect the sum of two floating-point numbers.

NOTE

If you do not want integers rounded down but you want them rounded to the nearest integer, use `Math.round ( )`*. The* `Math.ceil ( )` *function rounds numbers to the next highest integer, and* `Math.floor ( )` *rounds numbers down. The* `Math` *functions can be used with either numbers or strings.*

While `text` or `textarea` values are always in strings, changing them to numbers is quite easy, and JavaScript provides several functions to help you. However, the difficult part is remembering to do so!

Passing Data Between Forms and Variables

Before going on to discuss arrays, you need to understand the relationship that exists between HTML form properties and variables and values in JavaScript. You should understand something about passing data between forms and JavaScript variables. Consider the following form in an HTML page:

```
<html>
<body bgcolor=#BedFed>
<form name = "stateBird">
      <input type="text" name="state">
      <input type="text" name="bird">
</form>
</body>
</html>
```

The form itself is treated as an object in JavaScript, and, as was seen in previous chapters, objects are one of the data types that you can put into a JavaScript variable. The general format for using form data to a JavaScript variable is shown here:

```
var variableName = document.formName.propName.value;
```

For example, you can define the following JavaScript variables with the values of the contents of the form like this:

```
<script language="JavaScript">
var state=document.stateBird.state.value;
var bird=document.stateBird.bird.value;
</script>
```

Note that the element names in the form are being used as variable names. Now the object values are in variables using names of properties of the form object. The data values, however, are dependent on what a user puts into the text boxes.

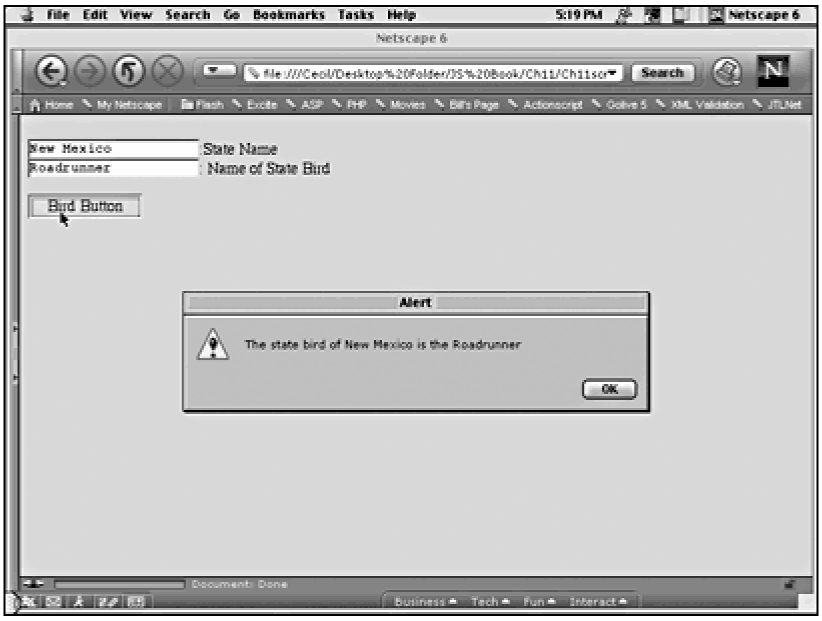
Putting these together along with a function and a button to fire off the function, you can see how the values of the variables change with output from a user.

```
formVariable.html
<html>
<head>
<script language="JavaScript">
function getBird( ){
      var state=document.stateBird.state.value;
      var bird=document.stateBird.bird.value;
      alert("The state bird of " + state + " is the " + bird);
      }
</script>
</head>
```

```
<body bgcolor=#BedFed>
<form name = "stateBird">
<input type="text" name="state">:State Name <br>
<input type="text" name="bird">: Name of State Bird<p>
<input type="button" value="Bird Button" onClick="getBird()";>
</form>
</body>
</html>
```

As you can see in Figure 11.2, the data entered into the form are echoed back in an alert window indicating the values in the forms were passed to JavaScript.

Figure 11.2. Data entered by users can be passed to variables in JavaScript.


File Edit View Search Go Bookmarks Tasks Help
5:19 PM
Netscape 6
file:///Ceol/Desktop%20Folder/JS%20Book/Ch11/Ch11scr
Search
Home My Netscape Flash Excite ASP PHP Movies Bill's Page Actionscript Golive 5 XML Validation JTLNet
Bird Button
Alert
OK
Document: Done
Business Tech Fun Interact

New Mexico State Name
Roadrunner Name of State Bird

The state bird of New Mexico is the Roadrunner

By the same token that values can be passed from a form to a JavaScript variable, the opposite is true. Data in JavaScript can be passed to a form using the following general format:

```
var variableName = someValue;
document.formName.propName.value = variableName;
```

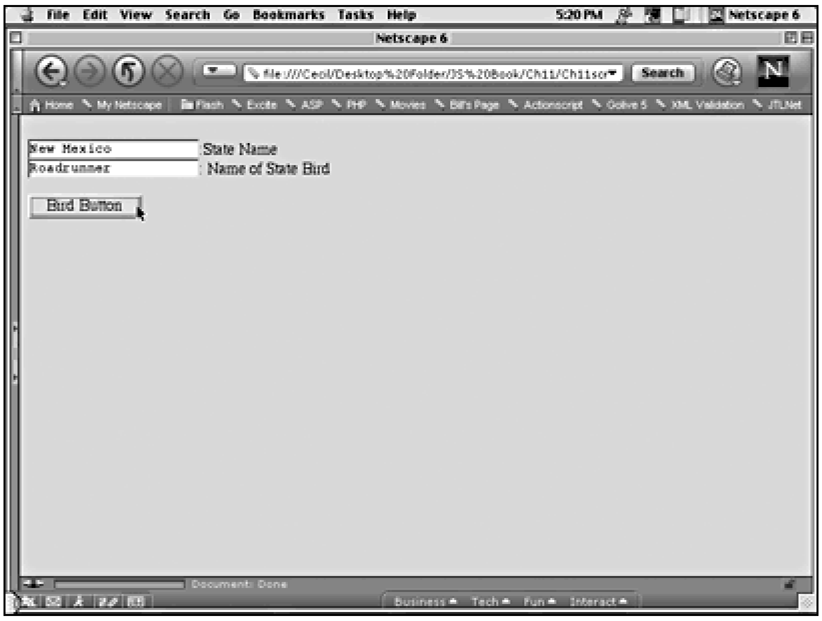
First, the JavaScript variable is declared and given a value. Then, the form object gets its value from the variable. The process simply reverses passing data from the form to the variable. The following program shows how to pass values from a variable to a form object in HTML.

variableForm.html

```
<html>
<head>
<script language="JavaScript">
function putBird( )  {
      var state="New Mexico";
      document.stateBird.state.value=state;
      var bird="Roadrunner";
      document.stateBird.bird.value=bird;
}
</script>
</head>
<body bgcolor=#BedFed>
<form name = "stateBird">
      <input type="text" name="state;">:State Name <br>
      <input type="text" name="bird">: Name of State Bird<p>
      <input type="button" value="Bird Button" onClick="putBird( )";>
</form>
</body>
</html>
```

Figure 11.3 shows what appears on the screen when the function fires. Until Bird Button is pressed, the two form windows are blank, indicating that the values in the JavaScript variables were indeed passed to the form object.

Figure 11.3. Data defined in JavaScript variables appears in the form windows.



New Mexico State Name
Roadrunner Name of State Bird

Forms as Arrays

When you create a form in HTML, you create a form array. The development of an HTML form has a parallel array in JavaScript. As each form is developed, a form element is added to the array. Likewise, as each element within the form is developed, another subelement is added to the array. All forms in a form array are properties of the document object and are referenced as `forms[n]`. Within each `forms[]` element, all of the elements added in the `<form>` container are elements of the `form[]` object. Table 11.1 shows the HTML and JavaScript parallels in developing an array.

Table 11.1. Parallel HTML and JavaScript Forms and Form Elements

HTML	JavaScript
`<form name="first">`	`document.forms[0];`
`<input type=text>`	`document.forms[0].elements[0];`

`<input type=button>`	`document.forms[0].elements[1];`
`<textarea></textarea>`	`document.forms[0].elements[2];`
`</form>`	
`<form name="second">`	`document.forms[1];`
`<input type=text>`	`document.forms[1].elements[0];`
`<input type=button>`	`document.forms[1].elements[1];`
`<textarea></textarea>`	`document.forms[1].elements[2];`
`</form>`	

Note in Table 11.1 that, like all arrays, the elements begin with `0` instead of `1`. Also note that, in the JavaScript for the second form, the `forms[]` element is equal to `1` but the first element of the `forms[1]` element is `0` because it is the *first* element of the second form. Also, note that `<textarea>` is treated the same as the `<input>` type of elements.

Forms and Elements

Because forms are numbered elements (properties) of the document object or form array, and because elements are numbered elements of the form array also, either or both of `forms[]` or `elements[]` can work with loops to enter or extract data in forms. By looping through a `forms[]` array, the form elements can be used in sequential order.

In the following script, a JavaScript function loops through the first of two forms. A button in the second form fires the function that sends the data to a text area that is part of the second form.

NOTE

The reference to the text area element is `document.forms[1].elements[1]`. *Because* `elements[1]` *is the second element in the second form, you might wonder why* `element[0]` *is not addressed. The reason lies in the fact that the first element in the second form is the button, not the text area. If you used* `element[0]`, *the text would attempt to stuff itself into the button.*

The script itself is fairly simple; however, because of all of the table tags required to format the script, a lot of the tag code is for the table elements.

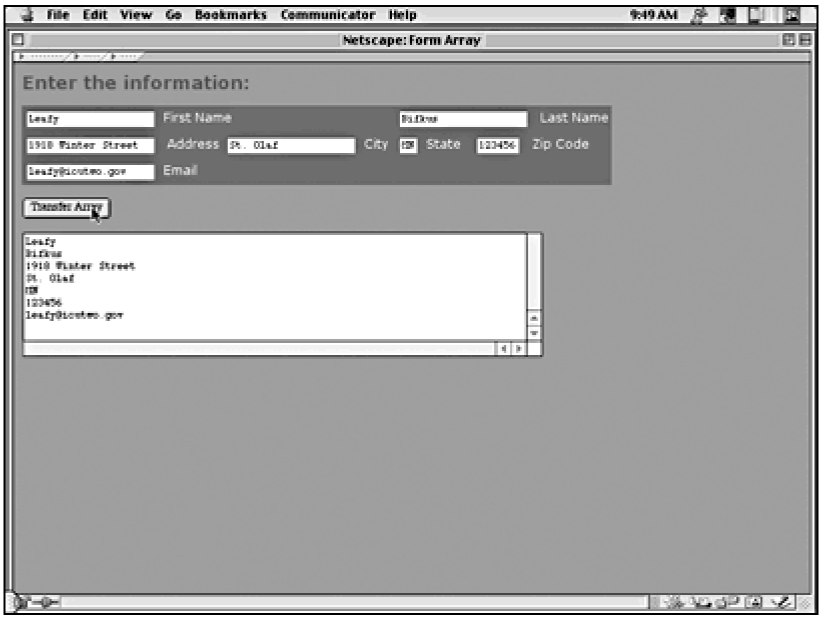
formArray.html

```
<html>
<head>
<title> Form Array </title>
<style type="text/css">
body {
     background-color:fe718 ;
     font-family:verdana
     }
.inBox {
     color:white;
     font-family:verdana
     }
```

```
h3 {
     font-size:18pt;
     color:a4352f
     }
</style>
<script language="JavaScript">
function sendEm( ) {
     var newDisplay="";
     var myForm=document.enter.length;
     for (var counter=0;counter < myForm;counter++) {
     newDisplay += document.forms[0].elements[counter].value + "\n";
     }
     document.forms[1].elements[1].value = newDisplay;
}
</script>
</head>
<body>
<h3>Enter the information:</h3>
<form name="enter">
<table border="0" cellpadding="3" cellspacing="0" height="78"
width="auto"
bgcolor="#8e58ad">
      <tr>
                        <td colspan="2"><input name="fName" type=text
><span
                        class="inBox">&nbsp;&nbsp;First Name</span>
</td>
                        <td colspan="2"><input name="lNamename"
type=text ><span
                        class="inBox">&nbsp;&nbsp; Last
Name</span></td>
                  </tr>
                  <tr>
                        <td><input name="address" type=text ><span
                        class="inBox">&nbsp;&nbsp;
Address</span></td>
                        <td><input name="city" type=text ><span
                        class="inBox">&nbsp;&nbsp;City</span> </td>
                        <td><input name="state" size = 2 type=text
><span
                        class="inBox">&nbsp;&nbsp;State</span></td>
                        <td><input name="zip" size=6 type=text ><span
                        class="inBox">&nbsp;&nbsp; Zip Code</span>
</td>
                  </tr>
                  <tr>
                        <td colspan="4"><input name="email" type=text
><span
                      class="inBox">&nbsp;&nbsp;Email</span> </td>
                  </tr>
      </table>
</form>
<form name="transfer">
      <input type=button value="Transfer Array"
onClick="sendEm( )"><p>
      <textarea rows=8 cols=80 name="gather"></textarea>
</form>
</body>
</html>
```

Because text windows have similar formatting requirements to alert boxes, the escape sequence `\n` is used to create a new line instead of `<br>`, as has been employed to create a line break when `document.write( )` is used to display output to a page. Figure 11.4 shows the output for the script.

Figure 11.4. You can use loops to extract and pass data between forms.



Enter the information:

Leafy	First Name			Bifkus	Last Name		
1910 Winter Street	Address	St. Olaf	City	MN	State	123456	Zip Code
leafy@icutwo.gov	Email						

```
Leafy
Bifkus
1910 Winter Street
St. Olaf
MN
123456
leafy@icutwo.gov
```

TIP

Even if you prefer to hand-code JavaScript to reduce code bloat found in the web site design applications, these tools (such as GoLive, Dreamweaver, and FrontPage) make it much easier to set up your tables. You can always make finer changes in the HTML and JavaScript code, but trying to design and format a page without some type of WYSIWYG (What You See Is What You Get—wizzy-wig) application is not a challenge that most designers need.

Addressing and Naming Forms

In the previous script, most of the references to the forms were to an array. However, you might have noticed that all of the elements except for the button had a name. Naming forms is a crucial step even if you address all elements as part of an array. Your script is much clearer if you have meaningful names assigned to all of the parts rather than array references. Using names makes everything much clearer. Table 11.2 shows some samples of well-named tags and the JavaScript reference to the form elements.

Table 11.2. Clear Reference Names

HTML	JavaScript
`<form name="addressBook">`	`document.addressBook;`
`<input type=text`	`document.addressBook.client.value`

`name="client">`	`="NanoNanoTech Inc.";`
`<input type=text name="lawyer">`	`document.addressBook.lawyer.value ="Sue A. Lot";`
`<form name="artSupplies">`	`document.artSupplies.length;`
`<input type=text name="brushes">`	`document.artSupplies.brushes.value ="Camel Hair";`

Not only do clearly named forms make debugging and updates a lot easier, but they also are essential for passing data to back-end sources. Beginning in Chapter 14, when JavaScript and different types of server-side programs are explored, you will find that the variable names passed to PHP, ASP, and Perl/CGI scripts are the names that you put into the HTML tags in forms. In cases when no server-side scripts exist in a web site, a smart designer still has named forms and form elements so that if a client wants a back end for a site, it's all ready to go.

Types of Forms

At the beginning of the chapter, you saw a list of different types of form attributes in HTML that JavaScript uses as sources of data input and extraction, for launching scripts, and for other action chores. This section of the chapter examines each element in the general categories established at the outset in more detail. Each of the element's JavaScript-relevant attributes are examined and discussed.

Input for Typing and Displaying

The most flexible interactive component on a web page is the text input element. As listed at the beginning of the chapter, five types of text input are found in HTML:

- `<input type=text>`
- `<textarea> </textarea>`
- `<input type=hidden>`
- `<input type=file>`
- `<input type=password>`

Throughout the book and this chapter, you have seen the input tags with the text type or the textarea containers. The hidden and password input tags work in a very similar way to input text, while the input file type is quite different.

Input Text

The input text tag has four important attributes relevant to both JavaScript and using forms:

- `name`
- `size`
- `maxlength`
- `value`

Of all of these attribute, `name` is the most important for identifying the element and its use as a variable name in server-side scripts. `size` refers to the size of the text window. The default size is 20, but you will have the opportunity to change the size to fit your needs. For example, as shown in the previous example, you need only two characters for state abbreviations and six for standard size ZIP codes. For street addresses, you might want 30 or 40 spaces available.

Related to size is the `maxlength` attribute. The `maxlength` attribute forces the user to a top limit. For example, if the user attempts to put in more than two characters in a text box set up for state abbreviations, she will find that it is impossible to do so.

Often, the `value` attribute is left out of a tag using input text. Usually, the page designer wants the user to put in her own information that will be used by the script. However, you can use default values in a text box as a prompt for the user. The following script shows a simple example of using all of the text box attributes:

```
<html>
<head>
<title> Text Box </title>
<script language="JavaScript">
function tooShort( ) {
      if (document.users.userName.value.length < 7) {
            alert("Your username must be at least 7 characters:");
      } else {
            alert("Your username is accepted:");
}
}
</script>
</head>
<body bgcolor="goldenrod">
<h4>Enter a user name between 7 and 12 characters long:</h4>
<form name="users">
      <input type=text name="userName" size=12 maxlength=12
value="username"><P>
      <input type=button value="Check user name"
onClick="tooShort( );">
</form>
</body>
</html>
```

Because the maximum size of `word` is restricted to 12 in the text attributes, using both `maxlength` and `size`, JavaScript does not have as much work to do. The function checks to see whether the length is correct and generates an alert window to announce whether the user name is acceptable. However, because the maximum length is handled by the HTML, all JavaScript has to do is to check for a length that is too short.

Textarea

The `textarea` container can be used as both an input and an output source of data in HTML and with JavaScript. Its key attributes are a bit different from those of the text box because *both* width and height must be included. The following four attributes are the most pertinent:

- `name`
- `cols`
- `rows`

- `value`

The following script shows how the value of the `textarea`, placed into an object, is filled with the wisdom of someone's boss:

```
<html>
<head>
<title> Text Area </title>
<script language="JavaScript">
function setUp( ) {
      var TA=new Object( );
      TA=document.memo.today;
            TA.value="JavaScript can be used with a variety of
applications and
            designs. This message shall not be altered by the viewer
on pain of a
            nose snuffle. Just try it! \n Signed, \n The boss"
      }
</script>
</head>
<body bgcolor="palegoldenrod">
<h3>Daily Memo:</h3>
<form name="memo"">
      <textarea cols=40 rows=8 name="today" readonly=0></textarea><p>
      <input type=button value="See Message" onClick="setUp( );"">
</form>
</body>
</html>
```

Hidden Text: Passing Unseen Data Between Pages

Hidden text boxes might not seem like too useful of a form element for designing web sites. However, hidden forms can be extremely useful when passing information between different pages in a frameset. To create a maze-like navigation system, used for anything from a cavern-and-tunnels adventure game to a problem-solving educational experience, knowing how to use JavaScript to simulate position is crucial.

The trick in any navigation system in which you have one driver page in a frame moving other pages left, right, forward, backward, or in any other combination of moves is to put the position information in the page that is being driven—not in the driver page. In this way, each page that is moved "knows where it is." That is, relative to any direction, you can put the information in the page that is going to be moved in any direction.

The next question is, "Where do you put the information on a page so that another page can use it?" The answer to that is, "Put it in a hidden form element!" To access form information in one page from another page in a frameset, use the following format:

```
parent.frame.document.form.element.value
```

Here, the element value is a URL. Then this statement will change the page in the frame to the value of the URL in the form element:

```
parent.frame.location = parent.frame.document.form.element.value
```

By hiding the information in a hidden form on the page being moved, nothing is visible to get in the way of your design. This next script is a bit involved but really very simple illustration of how hidden forms can be used to pass along information in a circular movement of pages going to the left or right. A frameset and common CSS style aids in holding it all together.

First, the CSS has a common set of body background color, center text alignment, font, and font size. Each of the pages has a different color font to help identify the page. The driver page has a background color added to the font color to make it stand out a bit more.

hidden.css

```
body {
      font-family:verdana;
      font-size:16pt;
      text-align:center;
      background-color:ffdb18
      }
.fontA {color:062456}
.fontB {color:f26d00}
.fontC {color:e43b24}
.fontD {color:188b57}
.fontDriver {
color:e43b24;
background-color:062456
}
```

The frameset page is important for naming the frame of primary reference. In this frameset, the target frame is the one named `info`. By removing all borders, the page appears to be a single unified page rather than a frameset.

hiddenSet.html

```
<html>
<head>
<title>Cross Page Communication</title>
</head>
<frameset rows="*,*" border=0 framespacing=0 frameborder=0 >
      <frame src="infoA.html" name="info" scrolling=no>
      <frame src="driver.html" name="navigate" >
</frameset>
</html>
```

The next step is to create a page that will have the navigation buttons. By using “left” and “right” directions only, the task is simple and focuses on the whole concept of moving one page with another by taking information from the page that is moved. However, you could add any combination of moves, including vertical as well as horizontal movement. The point is to use the information on a page, send it to another page, and then move the page with the information. It would seem to be a lot easier just to put the buttons on the page being changed and click buttons on that page. That is certainly true, but you would be unable to have a single navigational page that would not blink every time a new page is loaded. By using a single navigation page, not only do you have less coding, but you also would have a page that would unblinkingly stay in place while the other pages move. (You always get a little blink when one page loads and another unloads.)

The functions that make all of this work are quite simple. The function to move left is as follows:

```
var left=parent.info.document.position.hereL.value;
parent.info.location=left;
```

The function first places the hidden document element named `hereL` into a variable named `left`. Then the function uses the value of `left`, which now contains the URL for any left movement, and places it into the location of the frame that holds the page with the directional information. To see how this works, the following statement puts it into a single step using the array values instead of names:

```
parent.frames[0].location =
parent.frames[0].document.forms[0].elements[0].value;
```

In other words, the driver page tells the other page to find out where it goes next by specifying where it should look in itself—the hidden form elements. (Yoda would say, "The force is within you.")

driver.html

```
<html>
<head>
<link rel=stylesheet href="hidden.css">
<title>Driver</title>
<script language="Javascript">
function goLeft( ) {
      var left=parent.info.document.position.hereL.value;
      parent.info.location=left;
      }
function goRight( ) {
      var right=parent.info.document.position.hereR.value;
      parent.info.location=right;
      }
</script>
<body>
<div class=fontDriver>Click one of the buttons below</div>
<p>
<form name="direction">
      <input type=button value="<=Left" onClick="goLeft( )">
      <input type=button value="Right=>" onClick="goRight( )">
</form>
</body>
</html>
```

The final step is to have pages with the hidden forms. All four of the following pages are essentially the same, but each has a unique direction to go, depending on whether the left or right button is pressed in the driver page. If you wanted to have more directions, all you need to do is to add another hidden form with the information (URL) describing where to go.

infoA.html

```
<html>
<head>
<link rel=stylesheet href="hidden.css">
<title>InfoA</title>
<body><p>
<div class=fontA>
This is Page A<div>
<form name="position">
      <input type=hidden name="hereL" value="infoB.html">
      <input type=hidden name="hereR" value="infoD.html">
</form>
</body>
</html>
```

infoB.html

```
<html>
<head>
<link rel=stylesheet href="hidden.css">
<title>InfoB</title>
<body><p>
<div class=fontB>
This is Page B<div>
<form name="position">
      <input type=hidden name="hereL" value="infoC.html">
      <input type=hidden name="hereR" value="infoA.html">
</form>
</body>
</html>
```

infoC.html

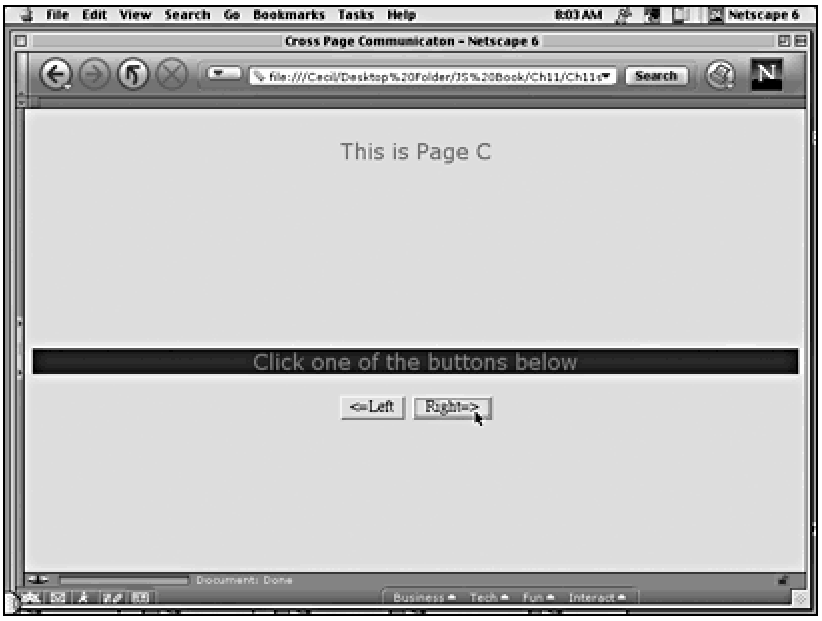
```
<html>
<head>
<link rel=stylesheet href="hidden.css">
<title>InfoC</title>
<body><p>
<div class=fontC>
This is Page C<div>
<form name="position">
      <input type=hidden name="hereL" value="infoD.htm">
      <input type=hidden name="hereR" value="infoB.html">
</form>
</body>
</html>
```

infoD.html

```
<html>
<head>
<link rel=stylesheet href="hidden.css">
<title>InfoD</title>
<body><p>
<div class=fontD>
This is Page D<div>
<form name="position">
      <input type=hidden name="hereL" value="infoA.html">
      <input type=hidden name="hereR" value="infoC.html">
</form>
</body>
</html>
```

Figure 11.5 shows how the frameset appears when the user clicks his way to Page C.

Figure 11.5. The page in the bottom frame navigates the page in the top frame by using information in hidden forms in the top page.


File Edit View Search Go Bookmarks Tasks Help
Cross Page Communicaton - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/Ch11/Ch11c
Search
This is Page C
Click one of the buttons below
<=Left
Right=>
Document: Done

Using hidden forms need not require assigning a value to the form. You can pass information to a hidden form just like you can a regular text box. However, in the previous example of using hidden forms, the purpose of the forms is to *give* information, not to *receive* it. For a very creative site, though, the information in the hidden forms could be dynamic, and, depending on circumstances in the context of the user clicking buttons, the information could change.

File Finder

Using file form elements is simple enough and can be useful in browsing files on local drives and disks. When you put in the tag `<input type=file>`, HTML automatically builds a Browse button and a text box. When the Browse button is clicked, an Open dialog box appears and you can select files from your own computer.

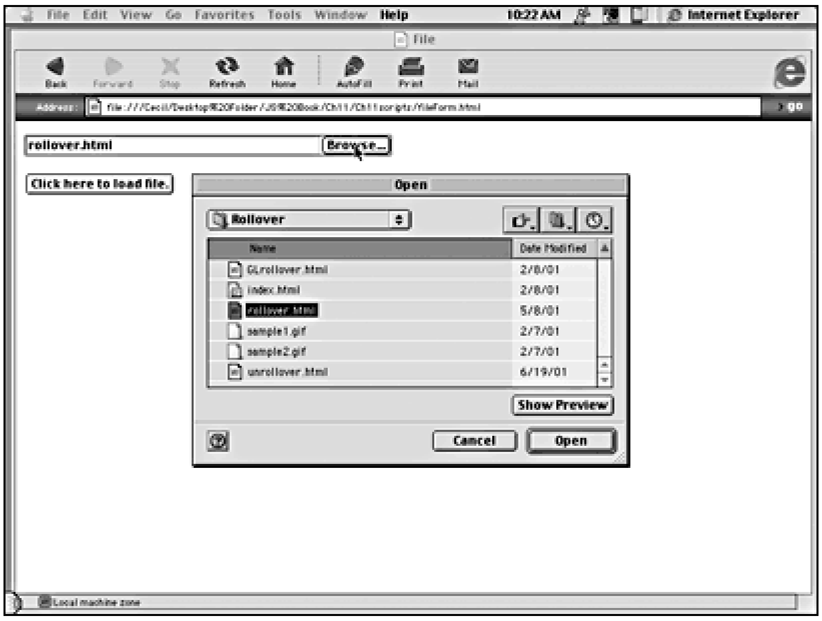
However, the purpose of the input text file form is to upload a file from the user's computer to a web server. In Chapter 16, "CGI and Perl ," you will see how to use CGI with input text files to upload files from your computer to a web server. The following simple utility to browse your drives and load a file shows you what the form generates:

```
<html>
<head>
<title>File Form</title>
<script language="Javascript">
function goGetIt( ) {
      window.location= document.seek.browseMe.value;
      }
```

```
</script>
<body>
<form name="seek">
      <input type=file name="browseMe" size=40 ><P>
      <input type=button value="Click here to load file."
onClick="goGetIt( );">
</form>
</body>
</html>
```

Figure 11.6 shows you what you will see when you use the input text file form.

Figure 11.6. The input text file form allows you to browse your files and upload pages to a web server.



NOTE

Netscape Navigator 6 users see the entire path on their desktop to a file, even if opened in the same directory as the web page browsing.

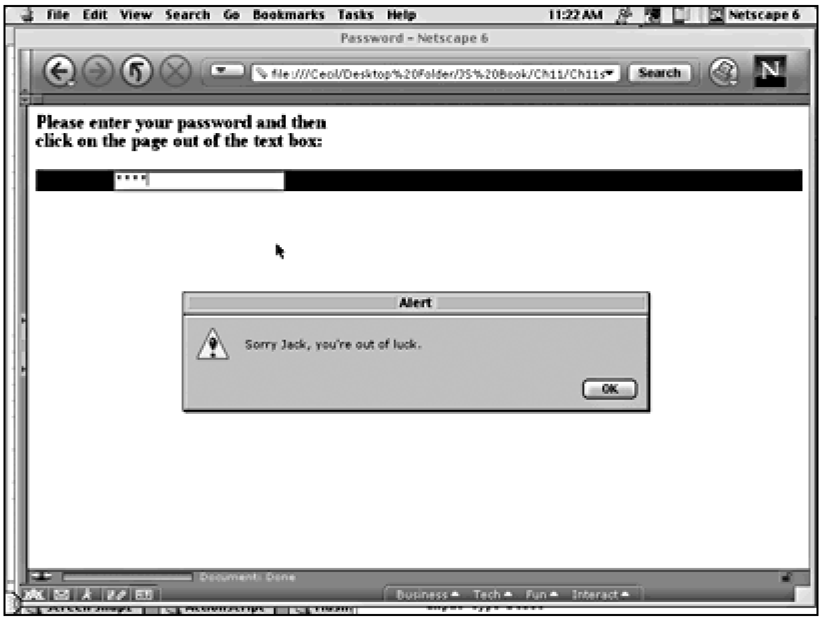
Password Form

The final type of input text form to be examined is the password form. It works just like an input text form, but no alphanumeric characters appear when the user types in her password. Instead, the user sees black dots. However, whatever is typed in a password form can be compared with alphanumeric characters in JavaScript in the same way that any value that is in a text box can be. The following example shows one typical use of the input password form element:

```
<html>
<head>
<style type="text/css">
.blackPatch {
      background-color:000000
      }
</style>
<title>Password</title>
<script language="Javascript">
function checkIt( ) {
var verify=document.pass.word.value;
if (verify=="JavaScript") {
      alert("You may pass.")
} else {
      alert("Sorry Jack, you\'re out of luck.")
      }
}
</script>
<body>
<h3>Please enter your password and then
<br> click on the page out of the text box:</h3>
<div class=blackPatch>
<form name="pass">
      some space <input type=password name="word"
onBlur="checkIt( );"> more space
</form>
</div>
</body>
</html>
```

If you want to hide the password from anyone who knows how to use view source on his browser, you can put the function in an external JavaScript file and hide it someplace where it cannot be easily found and viewed (see Figure 11.7). (For serious hiding, you will need something more secure than an external .js file.)

Figure 11.7. The password input box shows no alphanumeric characters, but JavaScript can use the input in the same way as an uncoded text box.


File Edit View Search Go Bookmarks Tasks Help
11:22 AM
Netscape 6
Password – Netscape 6
file:///Ceol/Desktop%20Folder/JS%20Book/Ch11/Ch11s
Search
Alert
OK
Document: Done
Business Tech Fun Interact

Please enter your password and then
click on the page out of the text box:

••••

Sorry Jack, you're out of luck.

Buttons and Their Events

Throughout the book, you have seen several examples of buttons. Usually, the buttons are employed to fire functions and sometimes forget that buttons are a form element instead of something else. The following are all of the HTML button elements to be discussed in this section:

- `<input type=button>`
- `<input type=reset>`
- `<input type=submit>`
- `<input type=radio>`
- `<input type=checkbox>`
- `<input type=image>`

The Generic Button

By this point in the book, you have probably seen most, if not all, of the attributes associated with the generic input button form. However, the following attributes are the most important to keep in mind:

- name
- value
- Mouse events

  `onClick`

  `onMouseOver`

  `onMouseMove`

  `onMouseDown`

  `onMouseUp`

  `onMouseOut`

  `onFocus`

  `onBlur`

Usually, `onFocus` and `onBlur` are associated with input text; however, both event handlers work with the button as well. The following shows a simple example. (You will need NN6+ or IE5+ for this next script.)

```
<html>
<body>
<form >
      <input type=button value="Blur Test" onBlur="alert('Blur
works')">
      <input type=button value="Focus Test" onFocus="alert('Focus
works')">
</form>
</body>
</html>
```

Most importantly for the button is the fact that it can call upon an event to fire a JavaScript function. The `value` attribute actually does store a value in a button, and the value can be changed dynamically. However, for most applications, the button value is simply used to identify what the button does.

Clearing Forms with the Reset Button

In previous chapters, you have seen the Reset button used to clear forms. However, the Reset button has the same attributes as other buttons, *in addition to* clearing forms. You can simultaneously launch a function using a Reset button while clearing forms. Enter the following script and open it in a browser window. Type some text into the text box and then click the button.

```
<html>
<body>
<form>
<input type=reset value="Click to Launch" onClick="alert('Function
launched')">
<input type=text>
</form>
</body>
</html>
```

Usually, the HTML that defines a Reset button is sufficient for most scripts; however, if the need arises to use the button to launch a function in addition to clearing the form, you can do so. The JavaScript `reset( )` method emulates the button's action, and if the generic Reset button generated by HTML does not fit into your design, you may elect to do so. Also, you *will* want to use the button's `value` attribute to better clarify what the button does (such as Clear Form and Start Over).

The Submit Button

In the chapters on using the server-side scripts (beginning with Chapter 14), you will use the Submit button to call on scripts running on the server. For example, the following script segment shows the relationship between the Submit button and the `<form>` tag in launching a server-side script:

```
<form name="storage" method=get action="showStuff.php"> //Script on
server to be launched
<input type=submit> //Effectively fires the action in the <form> tag.
```

You can use a `form.submit( )` script in JavaScript to launch a submit event and, design-wise, you might find it useful to do so. So, instead of having the generic Submit button generated by HTML, you can use any image or text style that you want as a button representation.

From the Submit button you can fire a simultaneous event in the `<form>` tag. Often used for form validation, data in the form will first be reviewed for accuracy before being sent to a database on a server. The `onSubmit` event handler activates when a Submit button belonging to the form is activated. For form validation using `onSubmit`, you need to write your JavaScript in relationship to a `true` or `false` outcome using the `return` keyword, as shown in the following simple script for form validation:

```
<html>
<head>
<title>Simple Validation</title>
<script language="JavaScript">
function CheckItOut(form) {
      var yourName=document.folks.info.value;
      if(yourName=="") {
            alert("You know your name don\'t you? \n So type it in!");
            return false;
      } else {
            alert("Thanks, " + yourName + " for sending in your
name.");
            return true;
      }
}
</script>
</head>
<body>
<form name="folks" onSubmit="return CheckItOut(this)" >
Your Name Please:
<input type=text name="info"><p>
<input type=submit value="Click to Validate and Submit" >
</form>
</body>
</html>
```

When using a function that returns `true` or `false` and when setting up your `onSubmit` event to expect a return of some sort, a `false` return effectively cancels the submit. By creating a submit cancellation, the user is allowed to fill in those portions of the form not filled in at all or filled in incorrectly (such as a missing "@" in an email address).

Radio and Check Box Buttons

The radio and check box buttons can be evaluated as being checked or not—a Boolean dichotomy. The key difference, and value, of the radio button is that only a single radio button with the same name can be checked. If you attempt to check more than one radio button with the same name, one already checked will pop to an unchecked position. With check box buttons, the user can check as many as she wants. Each button has three pertinent attributes:

- `name`
- `checked`
- `value`

The general format inside a form container of each button is shown here:

```
<input type=radio name="radName" value="someValue" checked=Boolean>
<input type=checkbox name="checkName" value="someValue"
checked=Boolean>
```

To be useful, you need to include both the `name` and the `value` attributes. Optionally, you can have the checked attribute included as a Boolean value of `true` or `false`.

Both the `checkbox` and `radio` objects can be used to launch JavaScript functions. The `onClick` event handler can be added to either a `checkbox` or a `radio` tag.

Read the checked status of either the check box or radio button in JavaScript through the checked property of either objects. A generic JavaScript conditional statement examining the checked status of a radio button would read as follows:

```
if(document.form.radio.checked==true) {
      var storage=document.form.radio.value
}
```

The `checked` property works something like a `value` property, except that it has only Boolean values. Because the buttons both have a value attribute, the buttons also have a `value` property in JavaScript, as the previous generic statement shows.

This following script provides an example of using the two different types of buttons. The `textarea` in the example script shows the returns of the values in the `checkbox` and `radio` button tags.

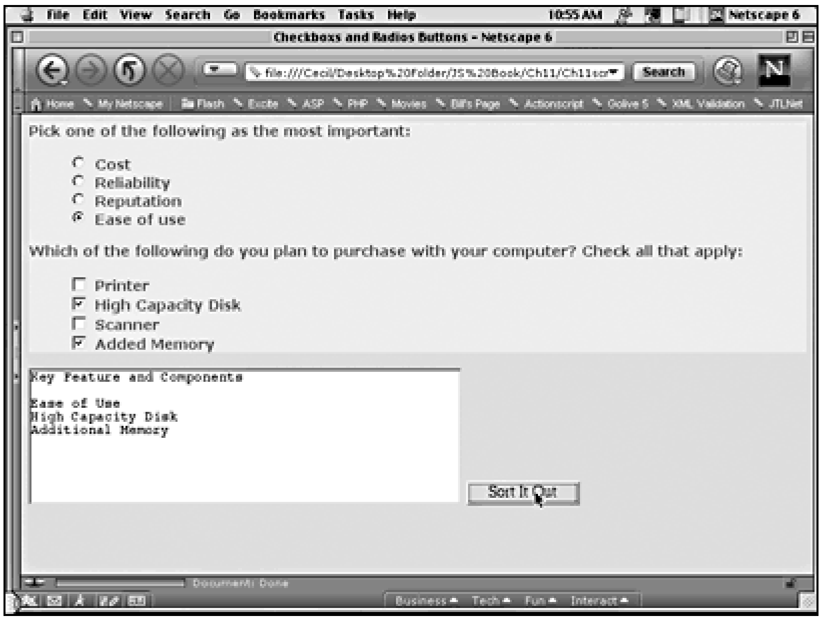
radioCheck.html

```
<html>
<head>
<title>Checkboxes and Radio Buttons</title>
<style type="text/css">
body {
      font-family:verdana;
      font-size:11pt;
      background-color:33ff66;
      font-weight:bold
}
.fStyle {
      color:595959;
      background-color:b3ffcc;
}
</style>
<script language="JavaScript">
function sortItOut(form) {
      var display="Key Feature and Components \n\n";
      var reviewer =new Object( );
      reviewer=document.hardware;
      for (var counter=0; counter < reviewer.length; counter++) {
      if(reviewer.elements[counter].checked) {
            display += reviewer.elements[counter].value + "\n";
            }
      }
      reviewer.showTime.value=display;
}
</script>
</head>
<body onLoad="document.hardware.reset( )">
<form name="hardware">
      <div class="fStyle">Pick one of the following as the most
important:<br>
      <ul>
      <input type=radio name="components" value="Cost" checked=true>
Cost<br>
```

```
      <input type=radio name="components" value="Reliability">
Reliability<br>
      <input type=radio name="components" value="Reputation"
      onClick="alert('Reputation?')"> Reputation<br>
      <input type=radio name="components" value="Ease of Use"> Ease
of use<p>
</ul>
Which of the following do you plan to purchase with your computer?
Check all that apply:<p>
<ul>
      <input type=checkbox name="printer" value="Printer" >
Printer<br>
      <input type=checkbox name="disk" value="High Capacity Disk">
High Capacity Disk<br>
      <input type=checkbox name="scan" value="Scanner" checked=true>
Scanner<br>
      <input type=checkbox name="mem" value="Additional Memory">
Added Memory<p></div>
</ul>
      <textarea cols=50 rows=8 name="showTime"></textArea>
      <input type=button value=" Sort It Out "
onClick="sortItOut(this.form)">
</form>
      </body>
      </html>
```

When you run the script, you will notice that some of the boxes are already checked because they were set as checked in the HTML tags. However, as soon as you click on a check box that has a default checked status or on an unchecked radio button, it becomes unchecked. (See Figure 11.8.)

Figure 11.8. Both radio buttons and check boxes can pass values through JavaScript.


File Edit View Search Go Bookmarks Tasks Help
Checkboxs and Radios Buttons - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/Ch11/Ch11scr
Search
Sort It Out
Document: Done

Pick one of the following as the most important:

- ( ) Cost
- ( ) Reliability
- ( ) Reputation
- (•) Ease of use

Which of the following do you plan to purchase with your computer? Check all that apply:

- [ ] Printer
- [x] High Capacity Disk
- [ ] Scanner
- [x] Added Memory

```
Key Feature and Components

Ease of Use
High Capacity Disk
Additional Memory
```

Image Form Elements

The input image is like a button object, except that the form element has a source image. The general format of the input image element is this:

```
<input type=image scr="imageURL" name="elementName">
```

The input image form works almost identically to using a graphic for links with the `<a href...>` tag. However, when using an input image element as part of a form, the element (object) is part of the forms object instead of the links object. Therefore, a reference to an input image object in JavaScript would be this:

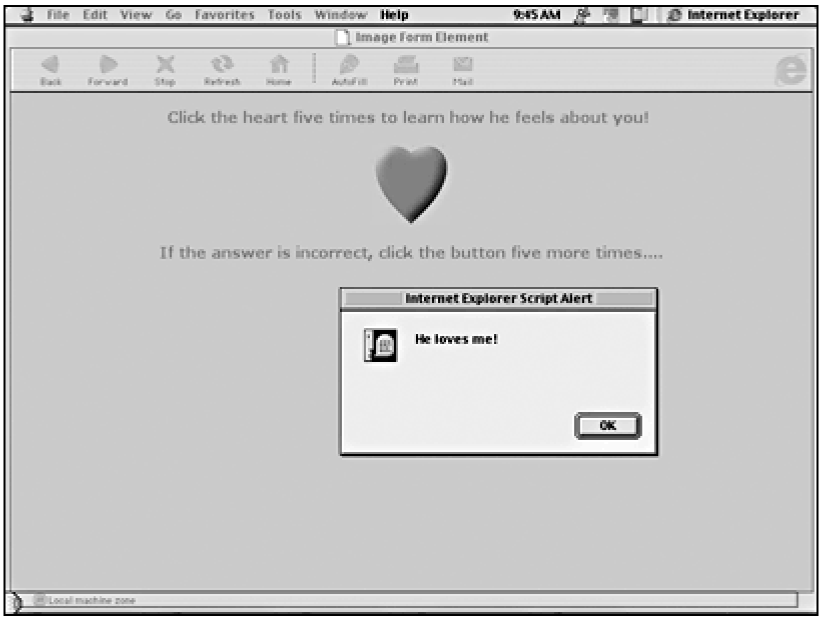
```
document.formsName.elementName
```

For designing web sites, having the capability to make your form buttons look like anything you want frees you from having to use the limited set of buttons built into HTML. The following script provides an example. (You will need to create a JPEG or GIF object named heart.jpg or heart.gif in the shape of a heart for a button.)

```
<html>
<head>
<style type="text/css">
body {
      font-family:verdana;
      font-size:11pt;
      color:red;
      background-color:#ffb7d0;
      font-weight:bold;
</style>
<title>Image Form Element</title>
<script language="Javascript">
function lovesMe( ) {
      var cupid=Math.random( );
      cupid=Math.floor(cupid*100);
      var throb="";
      if (cupid > 50) {
            throb="He loves me!"
            //Change the gender pronoun to your preferences
      } else {
            throb="He loves me not...."
            }
      alert(throb);
}
</script>
<body>
<center>
<form name="heart">
Click the heart five times to learn how he feels about you! <p>
      <input type=image src="heart.jpg" name="ache"
onClick="lovesMe( );"
       border=0><p>
If the answer is incorrect, click the button five more times....
</form>
</center>
</body>
</html>
```

Figure 11.9 shows the graphic button—a heart-shaped graphic.

Figure 11.9. Image buttons allow for far greater design freedom.



Click the heart five times to learn how he feels about you!

If the answer is incorrect, click the button five more times....

He loves me!

You may include rollovers on your input image button elements. In the previous script, for example, the reference to the pertinent form object in JavaScript would be as follows:

```
document.heart.ache.src=newImage.src;
```

In this way, the design can both contain a button that looks the way you want and has the liveliness of a rollover.

Menus

The last type of form element is the drop-down menu. The menu element in HTML is designed around three tags:

- `<select>`
- `<select multiple>`
- `<option>`

The `<select>` or `<select multiple>` containers contain any number of options. In a similar manner to radio buttons and check box buttons, respectively, the `<select>` tag is mutually exclusive and the `<select multiple>` tag is not. So, if the user has only a single choice, use the `<select>` tag; if he has multiple choices, use `<select multiple>`.

Each `<select>` container is an array element of the forms object, and each `<option>` is an array element of the `<select>` container. For example, consider the following menu set up with the `<select>` container:

```
<form name="menu">
      <select name="choose">
```

```
            <option> Randy
            <option> Judy
            <option> Fred
            </option>
      </select>
</form>
```

To reference the option Fred, the JavaScript array would read as follows:

```
document.forms[0].choose.options[2].text;
```

Note that the select object *must be* referenced by its name (choose), not an element[ ] number in the forms[0] array. Also note that Fred is a string literal in the text property, not the value property. The value property when dealing with select and option objects is the value assigned with in the <option> tag, such as this:

```
<option name="chooseMe" value="eternal wisdom"> Comedy
```

The following JavaScript statement would put the string literal eternal wisdom in the variable named stuff, assuming that the second option was Comedy:

```
var stuff = document.formName.selectName.options[1].value;
```

Most form elements in HTML have a value in the JavaScript object that reflects the corresponding HTML element. However, with the select object, the key value is called selectedIndex, reflecting which of the options the user selects. The selectedIndex keyword returns an array value for the options element. For example, consider the following HTML script and JavaScript reference to what is selected.

HTML

```
<form name="iceCream">
      <select name="flavors">
            <option> Chocolate
            <option> Strawberry
            <option> Vanilla
</option> </select></form>
```

JavaScript

```
var getChoice = document.iceCream.flavors;
var youChose = getChoice.options[getChoice.selectedIndex].text;
```

For example, if the user chooses Strawberry, the selectedIndex would return 1 because the second option is recognized as options[1]. Look at the following example to see what's going on under the hood in the page using a menu:

```
<html>
<head>
<title>Menu Form</title>
<script language="Javascript">
```

```
function getFlavor( ) {
      var scoop = document.iceCream.flavors;
      var cone = scoop.options[scoop.selectedIndex].text;
      if(cone !="Flavor Menu") {
            alert("Here\'s your " + cone + " ice cream cone!")
      }
}
</script>
<body onLoad="document.iceCream.reset( );">
<form name="iceCream">
<h3> Select your flavor from the menu:</h3>
      <select name="flavors" onChange = "getFlavor( )">
            <option name="dummy" value=null>Flavor Menu
            <option >Peach
            <option >Peppermint
            <option> Rasberry Swirl
            <option> Rum Raisin
            </option>
      </select><p>
</form>
</body>
</html>
```

The first option text is `Flavor Menu`. This text is added because, without it, you cannot effectively use the `onChange` event handler to select the choice at the top of the menu. Essentially, the option is a dummy one that allows a label inside the menu.

If you use the `<select multiple>` tag, the viewer can see all of the selection options at once. If you have room on your web page, using `<select multiple>` is usually easier for the user because she doesn't have to hold open the menu while making a choice. However, your own design considerations outweigh utility in this case. In the previous example script using `<select>`, change the tag to `<select multiple>` and run the script in your browser again.

Summary

Forms are one of the most important elements in HTML and JavaScript. With forms, the web designer can engage the viewer, provide feedback, and move information to different locations. The window boxes and buttons are both sources of data entered by the user and events that can be dynamically used in a page design. For purposes of discussing forms with design, they were categorized into *text, buttons,* and *menu* types of forms. Each of these form elements is further broken down into the different components. However, as JavaScript objects, most appear very much the same.

Menu forms are a bit different, but, when working with forms in JavaScript, each element still is a property of the document and the form. In turn, the attributes of the different types of form elements are properties, and all line up in similar JavaScript fashion as objects that can be used in very creative ways.

In the third section of this book, beginning with Chapter 14, forms are the doorway between HTML and server-side scripts and data. JavaScript is the doorkeeper for user-entered data and serves to verify data before passing it along to the back end. All of the data passed through the JavaScript filer, though, first appears in forms in some manner. So, while the next two chapters only marginally rely on JavaScript working with forms, subsequent chapters have a central role for forms and JavaScript.

Chapter 12. Dynamic HTML

CONTENTS>>

What Is Dynamic HTML?

With version 4 of both Internet Explorer and Netscape Navigator came Dynamic HTML (DHTML). At the heart of DHTML is the capacity to move objects dynamically on an HTML page and to use absolute positioning. Actually, absolute positioning and dynamic movement are related because changing one absolute position, or a position relative to an absolute one, to a different position simulates movement. To make DHTML work, something had to be available to trigger or drive the dynamic changes. For the most part, this role has been filled by JavaScript and event handlers.

In addition to the capability to put an object where you want it on a page without using tables or frames in HTML, another feature of DHTML is Cascading Style Sheets (CSS). As demonstrated in examples throughout the book, CSS is a designer’s best friend. Not only can all manner of font styles, types, sizes, and weight be controlled, but so can margins, indents, and other structural elements in text.

DHTML started the movement to separate style from content. In other words, HTML tables were never intended to control layout. It is much more ideal to control layout with one external style sheet.

About the same time that DHTML was introduced, programs such as Macromedia Dreamweaver and later Adobe GoLive made creating DHTML sites that much easier, albeit with some code bloat and hitches with cross-browser and cross-platform compatibility. However, the cross-browser incompatibility was not the application’s fault; it was the fault of Netscape and Microsoft for not adhering to a common standard developed by the World Wide Web Consortium (W3C) or working out one between themselves. To make a long and sordid story short, the incompatibilities between NN4 and IE4 were so great that developers were faced with creating sites for one browser or the other. Some attempted to write cross-browser scripts, but doing so literally doubled the work of the developer. It was easier to use animated GIFs movement and complex tables for approximating absolute positioning.

It is definitely a complex task to implement custom DHTML cross-browser/ platform programming. `If/Else` conditional statements, however, make it possible to write functions that will work on varying browsers/platforms.

NOTE

While the nonsense around DHTML was going on unresolved, developers and designers discovered Macromedia Flash. Not only did Flash do what they wanted as far as animation and absolute positioning was concerned, but it also did it at a very low bandwidth and was cross-everything–compatible with the necessary plug-in. Then both Microsoft and Netscape decided to embed the plug-in for Flash into their browsers, so the concern about users not having the appropriate plug-in was resolved. Instead of waiting for the two major browser developers to get their acts together with DHTML, many turned to Flash. (See Chapter 18, "Flash ActionScript and JavaScript," for a discussion of JavaScript's relationship to Flash.) With Flash 5 ActionScripting being based on JavaScript, it is becoming much more attractive to approach dynamic projects with Flash.

At the time of this writing, both Netscape Navigator and Internet Explorer were in Version 6 of their respective browsers on Windows platforms, and NN6 and IE5 on the Macintosh. While each still has its own way of positioning the contents of a `<DIV>` *container, some correspondence between the two is apparent in DHTML.*

Cascading Style Sheets

One relatively stable feature of DHTML is CSS. In looking at the two browsers and Windows and Macintosh operating systems, you find some differences, but not many. This is especially true with IE6 and NN6. In this section, I want to go over some of the more critical visual elements that are part of CSS. CSS has an aural component that is not addressed in this book, as well as many other features that make CSS an important designer's tool. One highly regarded source is *Cascading Style Sheets 2.0 Programmer's Reference,* by Eric A. Meyer (Osborne, 2001). Online, you cannot do better than the CSS2 standard, at http://www.w3.org/TR/REC-CSS2/about.html. Here, though, you need to understand something about the basics.

Standard Units of Measurement in CSS

The first feature—and one of the nicest—of CSS to be discussed is the units of measurement in CSS. Those with a design background, especially page design, are accustomed to working in a world in which measurement is in terms of pica, leading, kerning points, and similar units not available in standard HTML. However, with CSS, many of these familiar units of measurement are once again available for making a page. Table 12.1 shows the units of measurement available in HTML.

Table 12.1. CSS Standard Units of Measurement

Symbol	Meaning
em	Horizontal distance of the letter *m* relative to the point size of the current font
ex	Vertical height of the letter *x* relative to the current font
px	1 pixel unit on the computer monitor
in	1 inch
cm	1 centimeter
mm	1 millimeter
pt	1 point (1/72 inch, but actual size depends on monitor screen setting)
pc	1 pica is 12 points

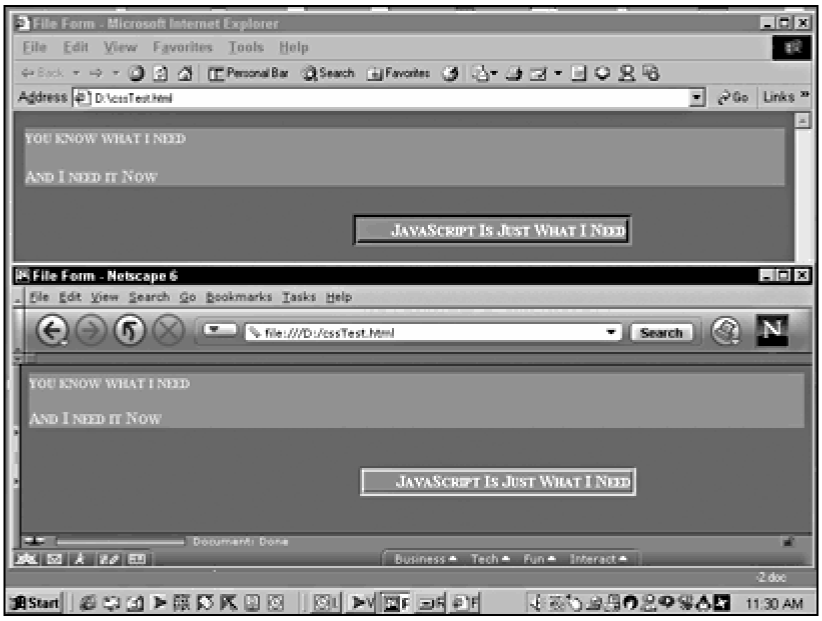
%	Percentage relative to another specified font size

Those without a page design background can find many books on page makeup whose principles of page design readily apply to HTML. Because of both the dynamic character of web pages, especially those with JavaScript, and the differing sizes of screens, you must design for web pages with a slightly different perspective. However, good page design for paper pages developed over the years almost always applies to web pages as well.

More Than Pretty Fonts and Colors

To get started, I want to provide a point of departure and reference. This next script has a little of everything in it and is shown on two different browsers (*NN6* and *IE6* on Windows) in Figure 12.1.

Figure 12.1. Only slight differences in the borders are noticeable when CSS is used in either NN6 or IE6.

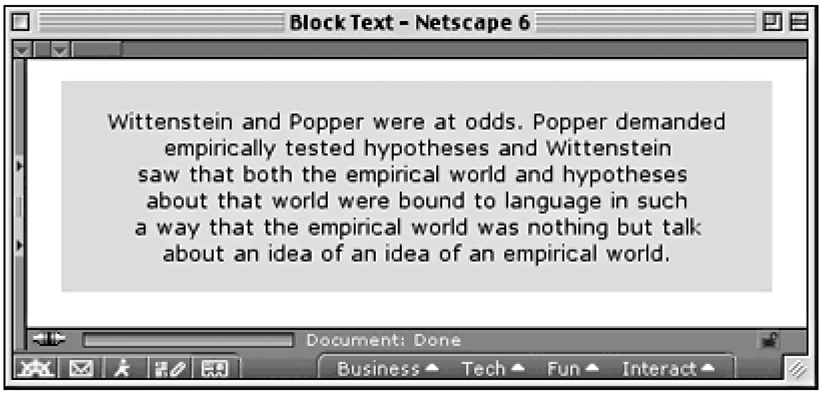


```
<html>
<head>
<title> CSS Potpourri </title>
<style type="text/css">
body {
      cursor:crosshair;
      background-color:ba3600;
      color:ebde33;
      font-weight:bold;
}
.test {
      border: groove;
      border-color:ab8f03;
      border-width:thick;
      position:absolute;
      left:300px;
      top:100px;
      text-transform:capitalize; //First letter of each word
capitalized
```

```
        font-family: verdana;
        background-color:ff4a00;
        color:white;
        margin-left:2em;
        text-indent:2em;
        text-align:center;
}
div {
        background-color:ab8f03;
        font-variant:small-caps;
}
</style>
</head>
<body>
<div>you know what i need
<span class="test" >
javaScript is just what i need
</span>
<p>And I need it Now</div>
</body>
</html>
```

Before going further, take a look at Figure 12.2 to see an approximation of what the screen appears like in the different browsers in a Windows environment.

Figure 12.2. Using the `<div>` tag helps to provide an integrated block of text.



Wittenstein and Popper were at odds. Popper demanded empirically tested hypotheses and Wittenstein saw that both the empirical world and hypotheses about that world were bound to language in such a way that the empirical world was nothing but talk about an idea of an idea of an empirical world.

If you look at the tags after the `<body>` tag in the script, you should notice that all of the text and other tags are within the `<div>` container—this text line:

```
you know what i need
```

followed by

```
javaScript is just what i need
```

and, finally:

```
And I need it Now
```

The first feature that you should notice is that the *second* line is below the first two. You might have expected the second line to be placed between the first and the third, but it is not because the second line is part of a `<span>` containing a CSS class definition (.text) that uses *absolute positioning.* So, even though all three lines of text are contained within the same `<div>` container, one of the lines is marching to the beat of a different CSS definition and positioning definition.

Look at the text, how it is formatted, the letter case in the script and on the screen, the different colors and their attendant relationship to the `<div>` or `<span>` containers, and the border and positioning attributes. The script will familiarize you with a cross-section of CSS. In the rest of the section, you will see how to use the major visual elements of CSS. Also run the script on pre-V6 browsers to see what is formatted and what is not. Version 4 browsers understand some of the CSS and not others.

<span> *and* <div> *Elements*

The `<span>` element is used in HTML with CSS to create an *inline* style. That is, only the materials within the `<span>` container are affected, and the rest of the paragraph is not. The `<span>` container causes no carriage returns or other breaks in the text. The `<div>` element, though, is a block element that will not flow in a paragraph. The `<div>` container can be used for absolute positioning of a block of text; within that block, other styles can be introduced using `<span>`. In the previous example, the `<span>` inside the `<div>` container took precedence over the `<div>` definitions but included them where they did not conflict. The following simple script shows a block of text within a `<div>` container:

```
<html>
<head>
<title>Block Text </title>
<style type="text/css">
div {
      font-family:verdana;
      font-size:10pt;
      background:#ddd;
      margin:1em;
      text-align:center;
}
</style>
</head>
<body>
<div>&nbsp; <br>Wittenstein and Popper were at odds. Popper
demanded<br>
empirically tested hypotheses and Wittenstein<br> saw that both the
empirical
world and hypotheses <br>about that world were bound to language in
such<br> a
way that the empirical world was nothing but talk<br> about an idea
of an idea of
an empirical world.<br>&nbsp;</div>
</body>
</html>
```

You will find creating blocks of text much easier using the `<div>` container than several `<span>` containers or user definitions. (You also might note that only three hexadecimal values were used in the background color definition. That’s a shortcut discussed further in the upcoming section "More on Colors.") Figure 12.2 shows the formatted output.

Tags and User-Defined Styles

As you have seen in many examples, HTML tags can be redefined to change the text and format characteristics using this format:

```
tag { attribute : value }
```

User-defined styles have two different formats. First, as you have seen, is the dot definition with the following format:

```
.name { attribute : value }
```

Second, you can define an ID style that is similar to a dot definition, except that it is prefaced by a pound (`#`) sign in the following format:

```
#name { attribute : value }
```

The only difference between the dot-defined style and the `ID` style, at the time of this writing, is that you apply the former using the `class` keyword and the latter using the `ID` keyword. In future developments of CSS, IDs are supposed to be unique (used only once on a page), and `class` can be used as many times as needed.

As has been seen in previous examples, the `class` keyword is used to select a given user-defined style. For example, if a dot-defined style were `.hiLite`, the class style selection would be this:

```
<p class=hiLite>...</p>
```

A selector defined by an ID (`#`) such as `#loLite` is applied using the `ID` keyword in the tag where it is applied, as the following shows:

```
<p ID=loLite>....</p>
```

The following script shows all three at work:

```
<html>
<head>
<title>ID and Class Text </title>
<style type="text/css">
body {
      font-family:verdana;
      font-size:10pt;
      }
.hiLite {
      background-color:yellow;
      color:blue;
      font-weight:bold;
      }
#loLite {
      background-color:blue;
      color:yellow;
      font-weight:bold;
      }
</style>
</head>
<body>
<span class=hiLite>Wittenstein</span> and <span
ID=loLite>Popper</span> never
resolved their differences. And <span class=hiLite>neither</span>
cared to.
</body>
</html>
```

Positioning in Three Dimensions

The absolute position properties in CSS include a location measured from the left side (x), the top (y), and the stack position (z). Generally, the position is measured in pixels (px), but you can use any of the legitimate units of measurement you want. The x and y positions will run in the hundreds of pixels, but the z position is relative to other `<div>` elements on the page. Like most other units in HTML and JavaScript, the `z` parameter begins with `0` for the lowest position on a stack of other `<div>` elements, to the highest number of `<div>` elements on the page. For example, the following position definition would place the objects in the `<div>` element about in the middle of an 800 × 600 pixel screen at level 2 in a stack of other `<div>` elements:

```
div {
      absolute: position;
      top:400px;
      left:300px;
      z-index:2;
      }
```

You must use the `<div>` element to use absolute position. You can use `ID` or `class` selectors to define a position; however, you then must place the selector into a `<div>` container. The following script shows how to position and stack layers using the `absolute position` keywords in CSS:

```
<html>
<head>
<title>Absolute Positioning</title>
<style type="text/css">
#top {
      background-color:rgb(130,146,20);
      color:black;
      font-weight:bold;
      font-size: 32pt;
      position: absolute;
      top:20px;
      left:20px;
      z-index:2;
      }
#middle {
      background-color:rgb(213,198,183);
      color:black;
      font-weight:bold;
      font-size: 32pt;
      position: absolute;
      top:50px;
      left:50px;
      z-index:1;
      }
#bottom {
      background-color:rgb(222,48,0);
      color:white;
      font-weight:bold;
      font-size: 32pt;
      position: absolute;
      top:80px;
      left:80px;
      z-index:0;
      }
</style>
</head>
<body>
<div id=top>At the top! </div>
<div id=middle>Caught in the middle! </div>
<div id=bottom>On the ground floor! </div>
</body>
</html>
```

You can also create a `relative position`. To have a relative position, you need a current position as a point of reference. So, if a `<div>` element is at 100,200 and a relative position of 50,100 is declared, the element would be moved to the position 150,300.

More on Colors

Up to this point, with a couple of exceptions in this chapter, I have used the six-character hexadecimal value or word colors (such as `lightseagreen` and `peru`) accepted by HTML to specify colors. However, CSS actually has four formats for colors:

```
.myColor { color: #rgb }
.myColor { color: #rrggbb }
.myColor { color: rgb(r,g,b) }
.myColor { color: rgb(r%, g%, b%) }
```

The first method duplicates each of the values in the `red`, `green`, and `blue` value slots. For example, this value:

```
#8ac
```

translates to

```
#88aacc
```

This technique can come in handy when adhering to a web-safe color palette. Any six-digit hexadecimal value with the following pairs of numbers is considered web-safe:

00 33 66 99 CC FF

Remembering 0, 3, 6, 9, C, and F is relatively easy; by stating a color as #39F, you know that you will be getting #3399FF, a web-safe color.

The third method, `rgb(d,d,d)`, shown in the script in the previous section, uses decimal values instead of hexadecimal ones. Some web art tools (such as Photoshop and Fireworks) use decimal RGB color values, and by copying those values into the formula, you can match a color exactly. Also, some books on color combinations show color values as RGB decimal ones. Leslie Cabarga's book *Designer's Guide to Global Color Combinations: 750 Color Formulas in CMYK and RGB from Around the World* (North Light Books, 2001) is a good example of a resource that can be used with the `rgb(d,d,d)` format. For example, the following color combination is from Tibet:

R	G	B	Description
238	222	233	Light plum
210	121	78	Brown
153	158	129	Gray-green
0	0	Black	

Using the colors from this palette, it is a simple manner to get the right values, as shown in the following script:

```
<html>
<head>
<title> RGB Colors </title>
<style type="text/css">
body { background-color: rgb(238,222,233);}
#header {
      background-color:rgb(210,121,78);
      color: rbg(0,0,0);
      font-family:verdana;
      font-size:18px;
      font-weight:bold;
      text-align:center;
      font-weight:bold
}
#bText {
      background-color:rgb(153,158,129);
      font-family:verdana;
      font-size:11pt;
      color: rgb(238,222,233);
      text-indent:2em;
      }
</style>
</head>
<body>
<div ID=header>Peace and Wisdom</div><p>
<div ID=bText>&nbsp;<p> Tibet is a culture with a rich spiritual
tradition.
<p> Someday it will be free again.<p>&nbsp; </div>
```

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

Finally, the fourth method of expressing colors in JavaScript is using the `rgb(%,%,%)` formula. Each of the values expects a percentage between 0 and 100. For those wanting to ensure a web-safe palette, this method is the easiest. Any of the following percentages in combination with one another is web-safe:

0% 20% 40% 60% 80% 100%

For example, having no clue what the following will present as a color, I know that the color is web-safe because each of the values except `0` can be evenly divided by 20:

```
rgb(80%,20%,60%)
```

Even if you do not require web-safe colors for your site, you will find that some color sources express colors as percentages of RGB, and it can be an easier way to envision the amount of red, green, and blue that you want in your "mixed" color.

Borders

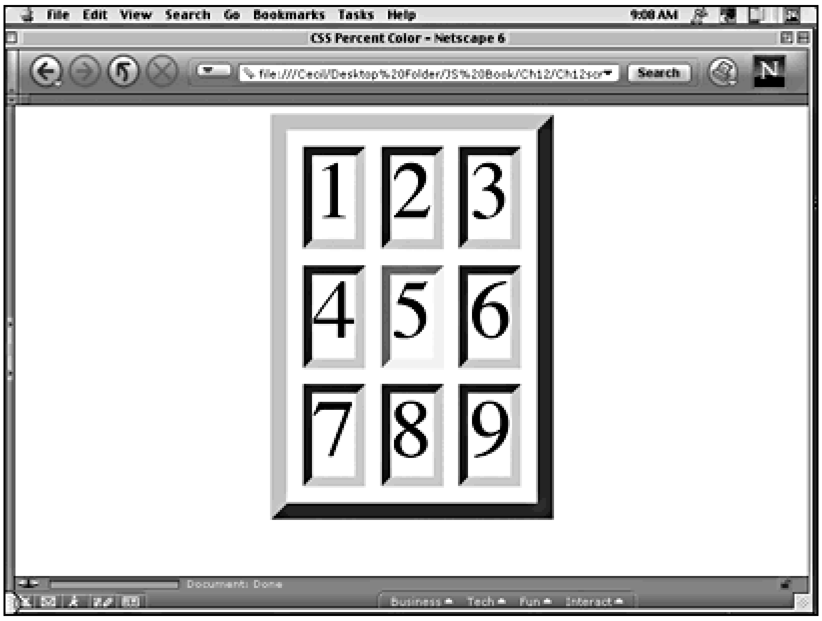
As a design element, borders are tricky calls. On one hand, borders represent one of the worse aspects of amateur design in terms of isolating, separating, and generally chopping up a page. On the other hand, if used judiciously, borders can sometimes act as useful or even subtle frames to evoke the right kind of attention to the information without calling attention to itself. CSS provides several border styles, as shown in Table 12.2 Each style is a value of the `border-style` property.

Table 12.2. Border Styles in CSS

Value	Effect
`none`	No border (my personal favorite).
`hidden`	Relevant only for using collapsing border model and inhibiting other borders. (Has precedence over all other styles.)
`dotted`	Border made up of dotted lines.
`dashed`	Border made up of dashed lines.
`solid`	Single line (default) makes borders.
`double`	Border made up of two solid lines.
`groove`	Border appears as indent on page.
`ridge`	Border appears to be raised from page.
`inset`	Entire border appears as groove in separate borders model.
`outset`	Entire border appears as ridge in separate borders model.

Besides `border-style`, you can set values for `border-width`, `border-spacing`, and `border-collapse`. Both the width and spacing values are expressed in terms of standard CSS units of measurement. The collapsing border model allows designers to specify all or part of a cell, row, row group, column, and column group. Borders can reference tables or independent blocks. The following script shows how various characteristics of borders are used in combination with size and color to create the table shown in Figure 12.3:

Figure 12.3. Fairly dramatic objects can be created at very little bandwidth costs using the border elements in CSS.


File Edit View Search Go Bookmarks Tasks Help
9:08 AM
CSS Percent Color - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/Ch12/Ch12scr
Search
1
2
3
4
5
6
7
8
9
Document: Done
Business Tech Fun Interact

```
<html>
<head>
<title> Borders </title>
<style type="text/css">
table {
      border: outset 12pt rgb(182,31,0);
      border-collapse: separate;
      border-spacing: 12pt;
      }
TD {
      border: inset 7pt rgb(80,101,38);
      font-size:60pt;
      }
TD.midcell {
      border: inset 7pt rgb(255,209,0) //Yellow in the middle
      }
</style>
</head>
<body>
<center>
<table>
      <tr>
            <td>1
            <td>2
            <td>3
      </tr>
      <tr>
            <td>4
            <td class="midcell">5 //Expect the yellow border
            <td>6
      </tr>
      <tr>
```

```
            <td>7
            <td>8
            <td>9
      </tr>
</table>
</center>
</body>
</html>
```

Text Formatting

One of the most valuable and underused formats on a web page is the indent. The elegant little notches in a stream of text serve to unify and separate. The indented text serves to demarcate one paragraph from the next, but not so much that the flow of ideas is divorced. The blocks of text that most pages contain have gaps that act as fissures in thought, made even worse by the obnoxious horizontal rule. Using CSS, putting in punctuation formats is simple. Table 12.3 shows the format for indents and various other text-related keywords.

Table 12.3. Text Formats in CSS

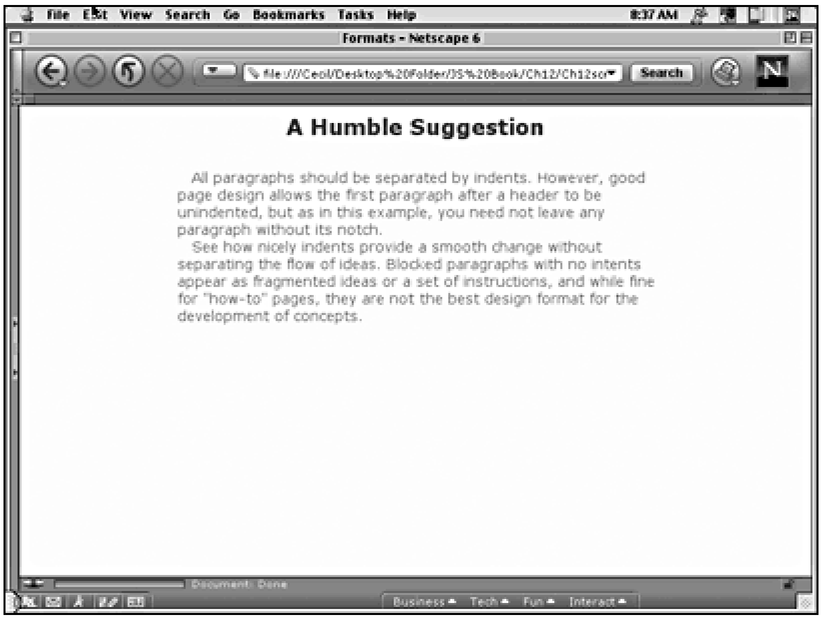
Format	Effect
`text-indent`	Indents first line of paragraph by measurement or percent
`text-align`	Uses left, right center, or justify
`text-decoration`	None, underline, overline, line-through, or blink
`text-shadow`	None, color, length (comma-separated: right, vertical below, blur radius)
`text-transform`	Capitalize, uppercase, lowercase, or none
`letter-spacing`	Normal, length
`word-spacing`	Normal, length
`margin`	Sets width for all margins
`margin-top`	Sets width of top margin
`margin-bottom`	Sets width of bottom margin
`margin-left`	Sets width of left margin
`margin-right`	Sets width of right margin

Cascading Style Sheets have more formatting options, and some, such as `text-shadow`, are not yet fully implemented. However, using the other formatting statements in CSS gives you control over your page’s appearance. Using “heavy” margins, the following example places the body text beneath the header without using any centering statements:

```
<html>
<head>
<title> Formats </title>
<style type="text/css">
#myBlock {
      margin-top:2em;
      margin-right:10em;
      margin-left:10em;
      text-align:left;
      text-indent:1em;
      text-transform:none;
      font-family: verdana;
      font-size:11pt;
      color: dimgray;
      font-weight:normal;
      }
#myHead {
      text-align:center;
      text-transform:capitalize;
      font-family: verdana;
      font-size:16pt;
      color: purple;
      font-weight:bold;
      }
</style>
</head>
<body bgcolor=lightyellow>
<div ID=myHead>
a humble suggestion </div>
<div ID=myBlock>
All paragraphs should be separated by indents. However, good page
design allows the
first paragraph after a header to be unindented, but as in this
example, you need
ot leave any paragraph without its notch.
<br><div style="text-indent:1em"> See how nicely indents provide a
smooth change without
separating the flow of ideas. Blocked paragraphs with no indents
appear
as fragmented ideas or a set of instructions, and while fine for
"how-to" pages,
they are not the best design format for the development of
concepts.</div>
</div>
</body>
</html>
```

One of the more difficult (and annoying) elements of the way CSS handles paragraphs and indents is that it adds an extra space using the `<p>` tag. To place a new paragraph without using the `<p>` tag, a `<br>` tag is used along with a `<div>` tag to define the indent. If a new `<div>` segment is set up and defined using the user-defined word `myBlock`, all of the text is reset using the margins of the initial `<div>` tag. In other words, it treats the material within the first `<div>` container as a layer, and the margins are measured from the sides of the layers, not sides of the screen. Figure 12.4 shows the output for the script.

Figure 12.4. With CSS formatting, you can get paragraphs to look like paragraphs.


File Edit View Search Go Bookmarks Tasks Help
8:37 AM
Formats - Netscape 6
file:///Ceol/Desktop%20Folder/JS%20Book/Ch12/Ch12sc
Search
Document: Done
Business Tech Fun Interact

A Humble Suggestion

All paragraphs should be separated by indents. However, good page design allows the first paragraph after a header to be unindented, but as in this example, you need not leave any paragraph without its notch.

See how nicely indents provide a smooth change without separating the flow of ideas. Blocked paragraphs with no intents appear as fragmented ideas or a set of instructions, and while fine for "how-to" pages, they are not the best design format for the development of concepts.

NOTE

The XHTML standard now requires that all tags be closed. One convention is to use `<br /><br />` *instead of* `<p>`. *This relates to "well-formed" pages that are XML-or XHTML-compliant.*

External CSS Style Sheets

When a CSS style sheet or a set of sheets for a web site is developed and refined, you can save a great deal of time by creating a CSS style sheet. As examples in previous chapters have shown, several different web pages can use the same style sheet saving time and bandwidth.

Follow this next set of steps to create an external CSS style sheet:

1. Write your styles as you would normally, except do not include any HTML tags other than those in CSS definitions.
2. Save the file as a text file with .css extension.
3. In the `<head>` container of your HTML page, enter this tag:
4.
```
<link rel="stylesheet" href="URL.css">
```

That's all there is to it. Design once; use often! The following two scripts show an example of a style sheet and a page that uses the sheet.

In the external sheet, note the absence of HTML tags to format the page. You can redefine HTML tags all you want, but don't use any in creating the style sheet.

external.css

```
body {
      margin:1in;
      background-color:rgb(249,230,158);
      }
#banner {
      text-align:center;
      text-transform:capitalize;
      font-family: verdana;
      font-size:24pt;
      color: rgb(159,183,138);
      background-color:rgb(115,113,73);
      }
.hottext {
      margin:1em;
      color:black;
      background-color:rgb(255,17,0);
      text-align:center;
      font-family: palatino,times;
      font-size:12pt;
      }
```

All references to the following HTML page are in relationship to the previous style sheet. Note the key CSS tag-connection line `<link>`.

external.html

```
<html>
<head>
<title> External </title>
<link rel="stylesheet" href="external.css">
</head>
<body>
<div ID="banner">
this banner is from a far and mysterious place
</div>
<div class="hottext">&nbsp<p>
This is just the kind of text I need for the whole site.
Thanks to my nifty
external style sheet, I don't have to re-do this style
every time I crank up my
Notepad or SimpleText to create some wild and crazy
JavaScript that uses CSS.
<p>&nbsp</p>
</div>
</body>
</html>
```

The bigger the site is, the more useful an external CSS file will be. *Any file* can be linked to a .css file, and its styles can be incorporated into your own site. (You can even tap into someone else's .css style sheet if you know the URL—only with the permission from the author, though.) An external style sheet works just like a web page as far as links are concerned.

The Role of JavaScript in Dynamic HTML

The most frustrating component of DHTML is the Tower of Babel that JavaScript has become in relationship to DHTML. With the fourth generation of both browsers, each (Netscape and Microsoft) uses a solution unique to its own browser. However, with the sixth generation, Netscape seems to have changed its mind about addressing CSS objects. So now, even if you go to all the work of creating multiple scripts to address the different browsers, their own internal consistency is suspect when it comes to JavaScript and CSS because the newer versions of the browser might not be compatible with the older version. Nevertheless, the future, while still the future, holds some promise of a merger. In the area of CSS, both NN6+ and IE5+, including IE6, use a common method of addressing IDs in CSS. The following section shows where this mutual point is and gives some brief history of where IE and NN have come from.

Netscape’s Solution

With the advent of NN4, JavaScript’s relationship to CSS and DHTML in general seemed to offer a clean solution to these new HTML objects. Basically, this format could assign a value or read a CSS object:

```
document.tags(ID/class).tagElement.property=value;
```

All of the attributes in HTML had to be redefined for JavaScript. For example, this line had to use the multicase word `backgroundColor` to replace `background-color` in CSS:

```
document.tags.body.backgroundColor="green";
```

The following script, which will work in NN4 but not NN6 or IE, illustrates this format:

```
<html>
<head>
<title> NN4 JS Style </title>
<style type="text/javascript">
      document.tags.body.backgroundColor="blue";
      document.tags.body.color="yellow";
      document.tags.body.fontSize="42pt";
      document.tags.body.fontFamily="verdana";
      document.tags.body.fontWeight="bold";
      document.tags.body.textAlign="center";
</style>
</head>
<body>
Big Bird Rules!
</body>
</html>
```

Unfortunately, the script not only is browser-specific, but it’s version-specific.

Microsoft’s Solution

Internet Explorer’s format is a bit different, and what works in IE4 also works in IE6. This general format can be used to assign a value to a given element’s property:

```
document.all.element.style.property="value";
```

For example, the CSS selector ID can be used to dynamically change an ID’s value. For instance, this line changes all instances of text using the CSS ID defined as `myID` to purple:

```
document.all.myID.style.color="purple";
```

Using JavaScript, you can change a lot more than just the color, as the following script shows:

```
<html>
<head>
<title> IE4 JS Style </title>
<style type="text/css">
#myFont {
      font-family:verdana;
      color: lightseagreen;
      font-size:32pt;
      text-align:center;
      }
</style>
<script language="JavaScript">
function turnPink( ) {
      document.all.myFont.style.color="pink";
      document.all.myFont.style.fontFamily="times";
      document.all.myFont.style.fontStyle="italic";
      document.all.myFont.style.fontSize="60pt";
      }
</script>
</head>
<body bgColor="dimgray">
<div ID="myFont" >Color Me Pink!<div><p>
<form>
      <input type=button value="Turn Pink" onClick="turnPink( );">
</form>
</body>
</html>
```

The New DOM Order

To play well together in the sandbox, NN6 and IE6 have adopted a far more object-oriented DOM based on the W3C model. In many ways, the new DOM resembles XML (see Chapter 17, “Working with XML and JavaScript”) and grows out of attempts by W3C to better integrate XML, HTML, CSS, and JavaScript. However, before donning party hats to celebrate the *détente* between the major browser providers, be aware that some differences might still exist. In other words, go slow on this new DOM until the full features of NN6 and IE6 come to light, and keep in mind that most browsers are still Versions 4 and 5.

The new DOM is a bit more demanding about containers, especially `<p>` tags, and for a good reason. Because the `<p>` tag can be used to select a CSS style, it needs both a `<p>` tag and a `</p>` tag to demarcate when to begin and end the selected style. More importantly, though, is that containers are a key type of *node* to be addressed in HTML by JavaScript. Elements that have no ending tag, such as `<br>`, or text on a page represent nodes as well. However, the containers are the key nodes because they divide the script into child and parent nodes. Parent nodes are containers that encompass another container node. Those encompassed nodes are called *child nodes.* For example, in the following script, you can see that the `<body>` node is inside the `<html>` node. Therefore, references to the parent node point to `<html>` container and the `<body>` container as the child node. The following HTML script shows an example of nodes with comments:

```
<html>                              Parent of Body
      <body>                        Child of HTML and parent of Form
            <form>                  Child of Body and Parent of Input
                  <input type=text> Child of Form and sibling of
Input
                  <input type=button> Child of Form and sibling of
Input
            </form>
      </body>
</html>
```

You can see five nodes in this HTML script. They include three container nodes and two independent nodes. The outermost tag is `<html>`, the parent of the `<body>` element. In turn, `<body>` is the parent of the `<form>` element, and the `<form>` element is the parent of the two `<input>` elements. The two `<input>` elements are siblings to one another because they are encapsulated in the same container, `<form>`.

One of the methods of the new W3C DOM is `getElementById( )`, and it begins to solve the mystery of why both classes and IDs are used in CSS. IDs have to be unique to be of any use when referenced. Otherwise, JavaScript would not know what part of a script is referenced when more than a single tag has the same ID; classes, on the other hand, can be used as much as you want in a script because they are not referenced by JavaScript using W3C DOM. The following script shows a sample of what the newer browsers will be capable of doing—and, *yes,* the script is cross-browser–compatible with IE5+/NN6+.

```
<html>
<head>
<title> The New DOM Order: NN6+/IE5+ </title>
<style type="text/css">
#brownOnBlack {
      font-family: verdana;
      color:peru;
      background-color:black;
      font-size:20;
      font-weight:bold;
      }
```

```
</style>
<script language="JavaScript">
function newDom( ) {
      var addTag=document.createElement("h1"); //Create new element
      var solution=document.createTextNode("The Brave New
DOM!");//New text
      addTag.appendChild(solution);//Put new text into new element
      document.body.appendChild(addTag);//Append the whole thing to
child
      }
</script>
</head>
<body bgColor="peru" >
<h1 ID="brownOnBlack">The Browsers are not getting along!</h1>
<form>
      <input type=button onClick="newDom( )" value="What is the
Solution?">
</form>
</body>
</html>
```

The heart of the W3C DOM approach to DHTML is very different from its predecessors. When you run the program, you will see the message "The Brave New DOM!" plastered on your screen in the default `<h1>` style. If you've ever tried to add text using `document.write( )` to a page with existing text, you will have found that it does not work. However, using this newer approach, you can make all kinds of dynamic changes that (cross your fingers) will run just fine on the new browsers from both Netscape and Microsoft.

To see how it works, you need to look at the JavaScript function line by line:

1. Because the program is going to add a new element, you need to create that element. It does not exist in the HTML page. The element selected is the `<h1>` tag, but you could have selected another element, such as `<p>` or any other HTML tag that contains text to be added.
2.
   ```
   var addTag=document.createElement("h1");
   ```
3. Next, you want to place the text for the newly created node into a variable. This variable will be used to specify the text node that will be appended to an existing node.
4.
   ```
   var solution=document.createTextNode("The Brave New DOM!");
   ```
5. Now use the `appendChild( )` method with the new element to identify the new element and what it is to do.
6.
   ```
   addTag.appendChild(solution);
   ```
7. Finally, you run down the hierarchy to pinpoint where you want to append the new child node in the HTML script. Hence, you need to use `appendChild()` a second time in the script. The top is `<html>` (`document`), `<body>` (`body`), and, finally, the child of `<body>` where the new material is placed.
8.
   ```
   document.body.appendChild(addTag);
   ```

At the time of this writing, IE6 is still in its infancy, and NN6 is going through growing pains as well. However, signs indicate that the future is very bright with the adoption of W3C DOM standards by both major browser manufacturers. The full integration brought about by having different elements of web scripting (XML, HTML, CSS, and JavaScript) adhere to a common DOM will add immeasurable value to JavaScript's utility as a dynamic tool in the future.

Summary

Whatever else you take away from this chapter, just remember that Cascading Style Sheets are the designer's best friend. This is true for several reasons. First, CSS provides the tools for designers to create pages that give the designer a good deal of control. Unlike non-CSS design, which relies on using a convoluted system of tables or using bandwidth-heavy graphics, CSS provides absolute positioning and a wide assortment of style, color, and background options. Second, CSS uses less bandwidth than scripts using graphics. Instead of relying on graphics, even ones boiled down to very low file sizes, CSS is nothing but cheap and light text instructions. Third, CSS is reusable. By employing external style sheets, a whole design palette can be reused after initial development. For real-world projects in which time is money, this feature alone sets CSS apart from other design solutions.

At this point in time and, for the next couple of years, designers using DHTML and JavaScript are going to have to either create multiple functions for cross-browser compatibility or hang back until both browsers have at least Version 6 installed in the bulk of the population. By adopting the Level 2 W3C DOM (and emerging Level 3 DOM), both Netscape and Microsoft have shown signs of maturity, optimistically pointing to a time when one design will be cross-platform– and cross-browser–compatible. In the meantime, learn all you can about the W3C DOM specifications at http://www.W3C.org. You will be glad you did as this new DOM comes to represent a truly universal language of the web.

Chapter 13. Remember with Cookies

CONTENTS>>

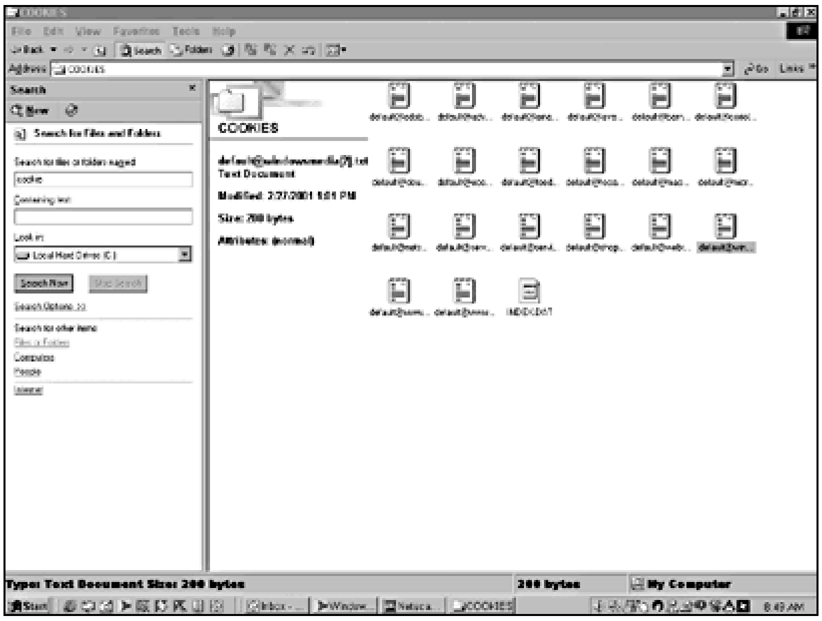
What Are Cookies and How Are They Used?

Think of cookies as lIttle lumps of data stored on the viewer's hard drive. Cookies are stored as data files in ASCII format, except on Macintosh, where you have a MagicCookie format. The text format precludes viruses being passed through them, and yet cookies provide a convenient and useful way of passing information about the user through the browser.

Cookies are used in a variety of ways. Usually, the designer wants the web page viewer to feel welcomed and focused on those key elements of personal interest. By examining the information on a cookie, the page can respond with the viewer's name and her personal favorites. For example, when I open my web portal page that I use for searches and news, I am greeted with "Welcome Bill!" Then the page shows all of the stocks, news stories, and other interests that I have. The way the page knows who I am is by reading my cookie file.

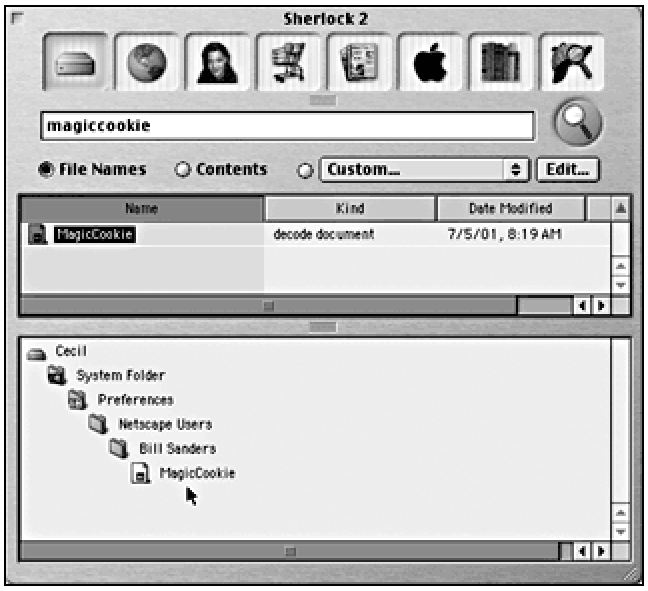
If you are concerned about personal information being used in the wrong way, a viewer can always delete his cookie file from his drive. If you have Windows OS, just select Start, Search, For Files or Folders. You will find a folder named COOKIES on your drive. You can throw every cookie or the entire folder in the Recycle Bin, if you want to delete them. Alternatively, you can just toss those cookies that you want removed. In my case, I don't want my cookies from the online bookstores deleted, so I keep them. On the other hand, I might have looked at a page with advertising that I don't want any more, so I can delete that cookie. Figure 13.1 shows a typical cookie folder's contents.

Figure 13.1. You can see the contents of all your cookies in text files stored on your Windows PC.



On Macintoshes, use Sherlock to search for MagicCookie. When you find it, you can look at the cookies using your word processor. Netscape stores your cookie file inside the System folder and subfolders within, as shown in Figure 13.2 Be careful if you make any changes to the MagicCookie file on your Mac using your word processor. It's easy to misalign the different cookies and mess up the ones that you want to keep. If you want to view or delete cookies on your Mac or PC using Netscape Navigator, open Edit, Preferences, Advanced, Cookies, View Stored Cookies. You can scroll through your cookies and view or delete individual ones. Using Internet Explorer, you can do the same thing by choosing Edit, Preferences, Receiving Files, Cookies.

Figure 13.2. On the Macintosh, Netscape stores cookies as MagicCookies.



Putting Cookies to Work

As far as JavaScript is concerned, cookies are document objects that can be used to read data through the viewer's browser and write data to the viewer's drive. When you open your browser, part of the document is a cookie; hence, this format can be used to extract or set a cookie:

```
document.cookie
```

When a page is loaded, this line places the cookie in the variable `varName`:

```
var varName=document.cookie;
```

The general format for setting a cookie is this:

```
document.cookie=name=value;[expires];[path];[domain];[secure];
```

The cookie's value is set as a name with a value. This format might be a little confusing because of a double assignment of name and value. However, when you get used to it, you can do some creative designs. The other attributes are optional, and they are discussed further in this section.

Creating a Cookie

In its simplest format, creating a cookie provides a name for the cookie and a value. For example, the following script sets a cookie with a name and a value:

```
<html>
<head>
<style>body {color:orangered} </style>
<script language="JavaScript">
document.cookie="sandlight=" + "Bill" + "*doughnuts";
</script>
```

```
</head>
<body>
<h1>The cookie is set! </h1>
</body>
</html>
```

The name used for the cookie is `sandlight=`, and I can look for that name later when I read the cookie. The value set for the cookie follows the name assignment after the equals sign. The value is a string, combined of `"Bill"` and `"*doughnuts"`. As you might have surmised, the combined value is really two values—the name (Bill) and something that this person likes (doughnuts). However, the asterisk (*) needs some explaining. You cannot place spaces, commas, or semicolons in cookie values. Because the value is a string, you can put in multiple values by using a unique character to separate the different values and then use JavaScript to sort things out. Alternatively, you can use the `escape( )` function to write data to a cookie and use `unescape( )` to extract data from a cookie.

When designing a site that uses cookies to store information about viewers (such as customers or potential customers), JavaScript's capability to work with sub-strings comes in handy. By using substrings and delimiters, such as the asterisk (*), you can add a good number of values to a single cookie. However, your cookies need to stay within 4K for the name and value. So, while you can be creative in their use, still keep names and values relatively small. Many designers use numeric codes instead of descriptive strings. For example, in designing a site for an online bakery, I might have 200 different bakery items. Instead of coding the different items using names (such as bearclaws, muffins, and doughnuts), I could have values from 000 to 199, each representing a different item. When a cookie has been established, on the next visit, the customer would be presented with the items he had previously indicated as favorites by referencing the coded items.

Reading Cookies

Reading a cookie's value is a matter of loading and parsing the cookie. With a simple application, such as the one established for writing a cookie, the job is not too daunting. The parsing work can be done with a `substring( )` function. The name, `sandlight=`, is 10 characters long (0–9), so the beginning of the value will be at 10. Hence, the statement to extract just the value would be this:

```
substring(10,cookie.length);
```

For example, the following script will extract the contents of the cookie created with the script in the previous section:

```
<html>
<head>
<style>body {color:orangered; font-size:24pt} </style>
<script language="JavaScript">
var myCookies=document.cookie;
var cookieVal=myCookies.substring(10,myCookies.length);
document.write(cookieVal);
</script>
</head>
```

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

When you launch the script in your browser, you will see the following on your screen in a nice orange-red color in 24-point serif font:

However, life with cookies is never quite so simple. In this next section, the discussion of optional attributes shows a bit more complexity in cookies.

Adding More Attributes

Cookies have the following four optional attributes:

- `expires` Cookie expiration date
- `path` Associated web pages
- `domain` Setting for multiple domains
- `secure` Boolean value for secure protocol

You can set some, all, or none of these attributes. Each attribute is placed in the order of this list, separated by semicolons. No labels identify the attributes, so you must remember the order and place dummy attributes if any are skipped.

Setting the Expiration Date

If `expires` is *not* set, your cookie is dropped at the end of the session; while it is optional, a cookie that evaporates at the end of the session where it is set is of little practical value. Therefore, most designers set the expiration date of cookies. Some cookies have relatively short expiration periods, but I have seen cookies set to expire in 20–30 years by optimistic designers.

Using JavaScript's built-in date objects, setting the `expires` attribute is quite simple. The date that goes into the actual cookie, though, must be in the `Date.toGMTString( )` format. For example, the following would set an expiration date to the date specified:

```
var cookieGone=new Date("December 26, 2004");
document.cookie="sandlight=" + "TimeSample" + "; expires=" +
cookieGone.
toGMTSring( );
```

Or, you can use a rougher setting, such as a couple of years:

```
var cookieGone=new Date( );
cookieGone.setFullYear(cookieGone.getFullYear( ) + 2);
var adios=cookieGone.toGMTString( );
document.cookie="sandlight=" + escape("Time Sample2") + "; expires="
+ adios;
```

The tricky part of setting the expiration is remembering that any label for the expiration date goes into quotes, along with the separating semicolon. In the two previous examples, look closely to how the `expires=` is encapsulated.

The Path

The best advice is to leave this setting alone. It automatically sets the path to the directory in which the page that generates the cookie resides. In cases where you want more than just those pages in the same directory to have access to a cookie, you can specify another path, such as /customers, and all pages within /customers will have access to the cookie. To open access to all web pages on the server, use / as the path. The format is as follows:

```
; path=/mypath
```

The Domain

When you want to override the access to a cookie to more than the server on which it is created, you can use the `domain` attribute. By specifying a domain name, such as .sandlight.com, pages on *any* server in the .sandlight.com domain can access the cookie. The path setting for this wide range of cookie access would have to be set to /. The format is as follows:

```
; domain=.myDomain.xxx
```

Note that a leading dot accompanies the domain name. Unless you write the entire URL beginning with `http://`, you need the leading dot.

Secure

If no `secure` setting is stated, the cookie is insecure. By typing the word `secure`, you make the cookie secure. The format is as follows, with no added values:

```
; secure
```

Getting Information and Giving It Back

The trick in using cookies, of course, is to have the user provide information and be able to get that information for use in the web page. Whether the information is for a simple greeting whenever the viewer opens her page or is used to configure the page being presented to the user, existing information must be placed in the cookie. This next script is a relatively simple one that asks the viewer for information that is stored in a cookie upon the press of a button. Then, by pressing a second button, the viewer is presented with *only* the value stored in the `name` attribute, but not the name itself or the date expiration information. Therefore, pay close attention to the formatting used for configuring input and output. (Remember that `&nbsp` is simply an HTML nonbreaking space.)

```
<html>
<head>
```

```
<title>Read and Write Cookie</title>
<style type="text/css">
      body {
      background-color:rgb(200,198,159);
      font-family:verdana;font-size:11pt }
      #display {color:rgb(169,33,53); background-
color:rgb(249,225,203) }
</style>
<script language="Javascript">
function yourCookie(name) {
      var remain=new Date("November 9, 2005");
      document.cookie= escape(name + document.baker.myCookie.value +
"; expires=" +
      remain.toGMTString( ) + ";");
      }
function welcomeBack( ) {
      var seeMe= unescape(document.cookie);
      var tagIt=seeMe.indexOf(";");
      seeMe=seeMe.substring(11,tagIt);
      alert("Hello there, " + seeMe)
      }
</script>
</head>
<body onLoad="document.baker.reset( )">
<div ID=display>&nbsp;
<form name="baker">
      &nbsp;<Input type=text Name="myCookie" ><br>&nbsp;
      Type in the value for your cookie and click the button to set
the cookie<BR>
      &nbsp;<input type=button value="Bake Cookie"
      onclick="yourCookie('goodCookie=');"><p>&nbsp;
      Click the button to read the value of the cookie along with a
greeting. <br>
      &nbsp;<input type=button value="See Cookie"
onclick="welcomeBack( );"><br>
      &nbsp;
</form>
&nbsp;</div>
</body>
</html>
```

The two functions provide a round-trip ticket for cookies. The first function sets the date and points the way to where the values can be found. This next line expects a name included when the function is placed in a tag to be launched:

```
function yourCookie(name)
```

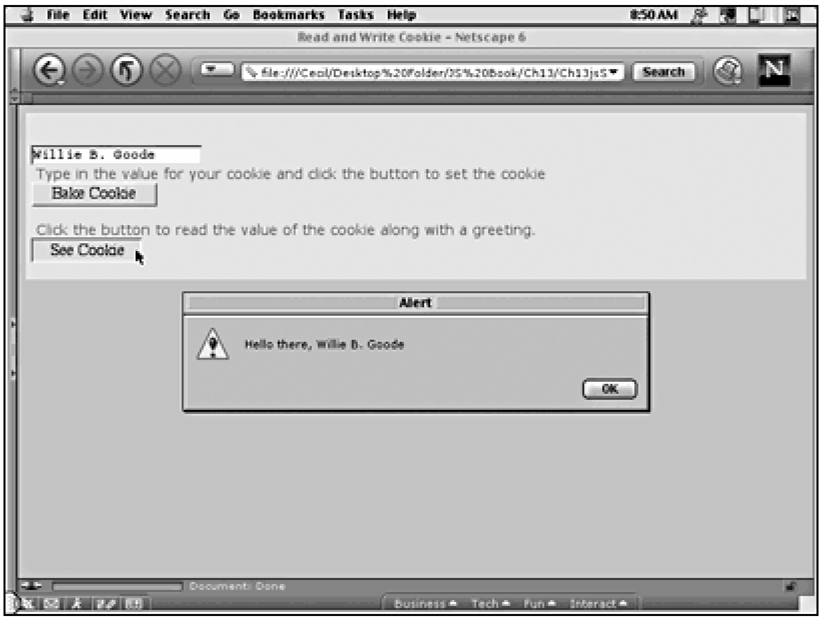
This line spells out where the information is to be found for the value (in the form) and sets up the expiration date:

```
document.cookie= escape(name + document.baker.myCookie.value + ";
expires=" +
remain.toGMTString( ) + ";");
```

Because the value for `name` is coming from a form, you can expect user input for the contents of the cookie.

The second function reads the cookie's value *only* and then puts the substring into a variable that is presented in an `alert()` function (see Figure 13.3). These lines first pull all the data from the cookie in the initial line, using the `unescape` function to decode it:

Figure 13.3. Users will generally be entering the valu es for cookies, and that information can be used in future visits to the site.


File Edit View Search Go Bookmarks Tasks Help
8:50 AM
Read and Write Cookie - Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/Ch13/Ch13js5
Search
Bake Cookie
See Cookie
Alert
OK
Document: Done
Business Tech Fun Interact

Willie B. Goode

Type in the value for your cookie and click the button to set the cookie

Click the button to read the value of the cookie along with a greeting.

Hello there, Willie B. Goode

```
var seeMe= unescape(document.cookie);
var tagIt=seeMe.indexOf(";");
seeMe=seeMe.substring(11,tagIt);
```

Then, using the `indexOf( )` method on the string containing the cookie, they locate the position of the semicolon that separates the value from the expiration date. Finally, the function looks for a substring beginning with the 12th character (remember, the string index begins with 0, not 1) because the name is 11 characters long. So, the substring between where the name (`goodCookie=`) ends and `expires` begins is the chunk of string where the value that you want can be found. (If you use a name other than `goodCookie=`, you will have to include the length of the name that you use as a starting point.)

Deleting Cookies

To delete a cookie, you need to provide it with an expiration date. It's quite simple because all you need to do is to set the same cookie name and then set an expiration date. The end user can always get rid of cookies set on her hard drive by throwing the cookie file into the Trash/Recycle Bin. A new cookie file is automatically regenerated when the browser is restarted.

Summary

Cookies can be used creatively to personalize a web site for visitors. The cookie information is not stored on a server, but rather on the user's hard drive; it represents the one thing that can be written to a user's disk from client-side scripts.

The designer's goal with cookies is to create a personalized environment for the user so that he will return to the site and keep using it. Cookies can be updated for changes in the site or the user's preferences. In the next several chapters, you will see how to store user information using server-side scripts and databases that are far more sophisticated and robust than cookies but that are used in a very similar way to cookies.

Part III: JavaScript and Other Applications and Languages

Chapter 14. Using PHP with JavaScript

CONTENTS>>

The PHP4 Scripting Language

For JavaScript users, PHP is an excellent transition language to server-side scripting. Now in Version 4 (or PHP4), PHP is officially named PHP Hypertext Preprocessor. The language was developed in 1994 by Rasmus Lerdorf as a server language for his "Personal Home Page," so he called it PHP. It is now a program scripting language with many similarities to JavaScript, so you should find it relatively simple to master.

PHP generally runs as a mix of HTML and PHP. PHP files are saved in the web server's root directory with the .php extension. The root directory varies with the setup of your system. If you are using your hosting service account, put your files into the root folder, and use your domain name as the root. (*Do not* put your PHP files in a cgi-bin directory.) For example, if your domain is `www.newriders.com`, and you save your PHP program in the root directory, you would enter this to call a PHP program:

```
http://www.newriders.com/yourProgram.php
```

Most designers put in additional directories for different projects to keep everything straight. For learning how to use JavaScript with PHP, you might want to add a directory in your web server root directory named PHP and put all of your PHP programs there. With the added directory, you would now enter this to access your PHP file:

```
http://www.newriders.com/PHP/yourProgram.php
```

The addressing process is identical to that of HTML files, but you have to keep in mind the relative relationship to the web server's root directory.

PHP Container

Like JavaScript, PHP code must be written within a container. PHP has three containers that you can choose from.

Container 1

```
<?php
script goes here
?>
```

Container 2

```
<?
script goes here
?>
```

Container 3

```
<script language="php">
script goes here
</script>
```

For this book, I will use Container 1. I like it because the beginning tag clarifies the fact that the script is PHP without a great deal of extra effort. The other two methods work fine, and, if you prefer, you can substitute one of them. (Note that Container 3 looks a lot like a JavaScript container.)

Writing and Testing PHP Script

Use a text editor such as Notepad (Windows) or SimpleText (Macintosh) to write your PHP scripts. If you use Notepad, when you save your file as a text file (text document) with a .php extension, make sure that Notepad doesn't add a .txt extension in addition to your .php extension. Place quotation marks around the name of your file when you save it, and Notepad will add no unwanted .txt extension. To get started, write the following script:

```
<?php
```

```
phpinfo( );
?>
```

Save the file as phpinfo.php in your server root directory or subdirectory. For this example, I've saved my PHP file in a folder (directory) named PHP in my web server's root directory, `www.newriders.com`. To launch the program, I would type this:

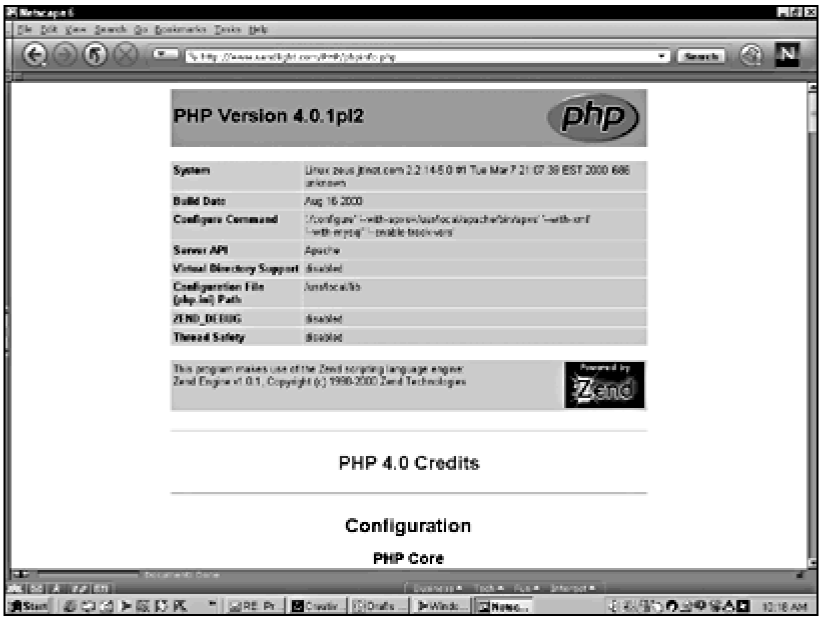
```
http://www.newriders.com/PHP/phpinfo.php
```

Substitute your domain name for `www.newriders.com`, and substitute your directory name for PHP. If you're using your computer as both a client and server, you would type the following, making the appropriate substitutions for the subdirectory name:

```
http://localhost/PHP/phpinfo.php
```

Figure 14.1 shows what you should see if everything is in the right place and is installed correctly.

Figure 14.1. What you see with a successful test of phpinfo.php.



PHP Version 4.0.1pl2

System	Linux zeus.jinot.com 2.2.14-5.0 #1 Tue Mar 7 21:07:39 EST 2000 686 unknown
Build Date	Aug 16 2000
Configure Command	'./configure' '--with-apxs=/usr/local/apache/bin/apxs' '--with-xml' '--with-mysql' '--enable-track-vars'
Server API	Apache
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/lib
ZEND_DEBUG	disabled
Thread Safety	disabled

This program makes use of the Zend scripting language engine:
Zend Engine v1.0.1, Copyright (c) 1998-2000 Zend Technologies

PHP 4.0 Credits

Configuration

PHP Core

If you see the page depicted in Figure 14.1 (or one close to it), you have correctly accessed PHP in your web server.

Beginning Formats

In PHP, you will see the `echo` command frequently in use. The command works something like a `print` command in Basic. The `echo` command takes the material to the right of the command and displays it in a web page. Generally, the `echo` command uses this format:

```
echo "Text, literals or variables.";
```

You can also use the `echo` command to show the output of functions, as in the following:

```
$fruit="BANANAS";
echo strtolower($fruit);
```

The output would display `bananas` because the function changes all characters in a string to lower case. Throughout the rest of this chapter, you will see the `echo` command used often because it is the workhorse command to display information to the user through web pages. However, the `echo` command can also be used to send data back to the server. (The `print` statement is also used instead of `echo`, and if you see a PHP listing with `print`, it is likely to be the developer's preference over using `echo`.)

Unlike JavaScript, PHP demands a semicolon at the end of most lines. If you leave a semicolon out of certain PHP lines, the program will not execute. For example, a simple variable definition like this one requires that the variable declaration itself end with a semicolon at the end of the line:

```
<?php
$fruit="Kiwi Fruit";
?>
```

The semicolon is the instructor terminator for statements. However, as in JavaScript, conditional statements never have semicolons after a curly brace. The following shows where the semicolons go and do not go:

```
<?php
if ($price => $sale) {
      $newPrice = $price - ($price * .1);
      }
?>
```

PHP is very unforgiving when it comes to semicolons at the end of a command line, so keep an eye out to make sure that your script contains the semicolons that it needs. When debugging your script, first check to see if your semicolons are where they belong. (Your practice of putting in semicolons in JavaScript should really pay off here.)

Comments

As in JavaScript, comments in your code help you remember what you're doing with the code. I will be using the double forward slash (`//`), just like in JavaScript. The following shows the correct use of comments in a PHP script:

```
<?php
$newName = $town . ", " . $state;
//The town and state are joined with a comma between them.
echo $newName;
```

```
?>
```

The comment line is another exception to the semicolon requirement. You can write anything in a comment line because, as soon as the parser sees the two slashes, it ignores the entire line. You might also see comments written over several lines that begin with `/*` and end with `*/`. For example, the following is a multiple-line comment:

```
/* The following comments are made to help clarify how
multiple line comments can be made in a PHP script */
```

Escape Characters

You will find the escape characters in PHP identical to those in JavaScript. A reverse slash (\) escapes the character's usual function. For example, the following shows how to include quotation marks in a variable:

```
<?php
$favoriteQuote = "Roosevelt said, /"There is nothing to fear but fear
itself./"";
echo $favoriteQuote;
?>
```

When the program is executed, the viewer sees the following on the screen:

```
Roosevelt said, "There is nothing to fear but fear itself."
```

Initially, the only character that you will have to remember to escape is the double quotation mark ("). However, when preparing data for a MySQL database, other characters need to be escaped, such as the single quotation mark (') and the ampersand (`&`). PHP can handle either character without escaping them, but the database cannot. PHP's `addslashes( )` function will help take care of the problem by adding the needed slashes automatically when the PHP script is used to add data to a MySQL database. Another function, `stripslashes( )`, can then be used to remove the slashes so that they won't cause a parsing problem with PHP.

Variables

Declaring a variable in PHP is not unlike doing so in JavaScript, except that all variables in PHP are preceded by a dollar sign (`$`), and you don't need the `var` statement. The following script shows integers, floating-point numbers (doubles), and string literals entered into PHP variables:

```
<?php
$enrollment = 25;
$itemCost=390.21;
$goodAdvice= "Remember Pearl Bailey.";
echo "<p>$enrollment";
echo "<p>$itemCost";
echo "<p>$goodAdvice";
?>
```

The output of the previous example follows:

```
25
390.21
Remember Pearl Bailey.
```

The `<p>` tag simply places an HTML paragraph between each variable output. For the most part, when dealing with PHP and JavaScript, the formatting is accomplished by placing the output in the appropriate variable. What is viewed on the screen will depend on whether you are placing the output from PHP into a variable that will be placed into a form or used in conjunction with `document.write( )` or some other statement that places variable information on the screen.

Operators and Conditional Statements

Most of the operators in JavaScript and PHP are the same. Table 14.1 shows the operators used in PHP.

Table 14.1. PHP Operators

Operator	Use	Format Example
=	Assignment	`$inven =832;`
+	Addition	`$total = $item + $tax + $shipping;`
-	Subtraction	`$discount = $regPrice - $salePrice;`
*	Multiplication	`$itemTotal = $item * $units;`
/	Division	`$distrib = $all / $part;`
. (dot)	Concatenation	`$location = $city . ", " . $state;`
%	Modulus	`$remainder = 85 % 6;`
==	Equal to (compare)	`if ($quarter1 == $quarter2) {`
===	Equal to + data type	`if($forest === $trees) {`
!=	Not equal to	`if (999 != $little) {`
>	Greater than	`if ($elephant > $mouse) {`
<	Less than	`if ($subTotal < 85) {`
>=	Greater than or equal to	`if ($counter >= 200) {`
<=	Less than or equal to	`if (300 <= $fullAmount) {`
+=	Compound assign add	`$total += 21;`
-=	Compound assign subtract	`$discount -= (.20 * $item);`
.=	Compound concatenation	`$areaCode .= $phone`
&&	Logical AND	`if ($first == 97 && $second <=` `$third) {`
||	Logical OR	`if ($high === 22 || $low == 12){`
++	Increment (pre or post)	`for ($la=6; $la <=78; ++$la)`
—	Decrement (pre or post)	`for ($ls=50; $ls >=12; $ls—)`

JavaScript concatenates strings using the plus (+) operator, and PHP4 uses the dot (.) operator.

Conditional Statements

The conditional statements in PHP4 are very similar to those in JavaScript but are not identical. PHP4 supports three different statements for branching a script.

The if *Statement*

PHP4 uses the `if` statement in the same way as JavaScript. When the condition in parentheses is met, the program executes the next line. If the condition is not met, the code in the next line is ignored. For example, the following code will trigger the `echo` line:

```
<?php
$cost=55;
$retail=70;
if ($cost < $retail) {
      echo "You can make a profit.";
      }
?>
```

The else *Statement*

When you have an alternative to the conditional statement using `else`, the `else` clause is initiated automatically when the `if` condition is *not* met. The following script shows how this works:

```
<?php
$designTime=88;
$payment=5500;
if ($payment>5000 && $designTime<80) {
      echo "Take the job";
      } else {
      echo "The schedule is too long.";
      }
?>
```

Note in the previous script how the logical `AND` (`&&`) operator is used in the script. Both PHP4 and JavaScript use the logical operators the same way.

The elseif *Statement*

The formatting of `elseif` in PHP4 and JavaScript is slightly different. PHP4 uses the single term, `elseif`, while JavaScript uses `else if` as two words. However, the rest of the formatting is similar. The following example illustrates how an `elseif` statement is set up in PHP4:

```
<?php
$designTime=88;
$payment=5500;
if ($payment>5000 && $designTime<80) {
      echo "Take the job";
      }
```

```
elseif ($payment>6000 || $designTime<90) {
echo "This deal will work";
}else {
      echo "We just cannot make this work";
}
?>
```

When using elseif statements, remember that you need an if statement before beginning a series of elseif statements, and you need an else statement after the last elseif statement, just like in JavaScript. Conditionals are the decision makers in PHP, as in JavaScript, and when creating a dynamic site, the data from the JavaScript in the HTML front end often must be analyzed in a PHP script.

Loops

PHP has three types of loops, and they are structured very similarly to JavaScript loops.

for Loop

A beginning value, a termination condition, and the counter (index) control the for loop. Increments and decrements in the index counter can come before or after the index variable. If the increments/decrements are before the index variable, the change occurs before the counter is employed, while increments/ decrements after the index variable generate change after the going though the loop.

```
<?php
for ($counter=0; $counter <100; ++$counter) {
      echo $unit . $counter;
}
?>
```

while Loop

A statement at the beginning of the while loop stipulates the conditions under which the loop terminates. All loop actions take place between the curly braces and include an incremental or decremental variable.

```
<?php
$counter=100;
while ($counter >1) {
echo $unit . $counter . "<BR>";
//$unit is a variable passed from HTML page using JavaScript$counter–;
}
?>
```

do...while Loop

Because the counter is at the bottom of the do...while loop, the loop must be processed at least once. The while loop can be terminated before any statement is executed.

```
<?php
$counter=12;
do {
echo $unit . $counter . "<BR>";
//$unit is a variable passed from HTML page using JavaScript$counter—;
} while ($counter >1);
?>
```

Arrays

Arrays work the same in both JavaScript and PHP, and the differences are minor. You will find arrays an important PHP tool to use when pulling data from a database and passing the information to JavaScript. Note the differences between array construction in JavaScript and PHP as well as the several ways that arrays are constructed in PHP:

```
<?php
$book[0] = "The Odyssey";
$book[1] = "The Iliad";
$book[2] = "The Republic";
$book[3] = "Philosophical Investigations";
$book[4] = "Heaving Romances!";
?>
```

Like arrays in other scripting and programming languages, the initial element is the `0` element, and not `1`. If you list array elements with only the brackets and no numbers, PHP automatically assigns each element based on order of assignment, beginning with `0`.

You can also use the array constructor to build an array. It works very much like JavaScript’s constructor. For example, the following is an array of fruit:

```
<?php
$fruits = array ("peach", "apple", "plum", "orange");
?>
```

In referencing the array elements, the first value in the array is a peach and is assigned an element identifier of `0`. The others are assigned element numbers sequentially so that orange would be element `3`. The following script shows the peach and the plum:

```
<?php
echo $fruits[0];
echo $fruits[2];
?>
```

An excellent array feature of PHP is the capability to set the index. A special array operator, `=>`, sets the sequence. For example, if you want your `$fruits` array to begin with `1` instead of `0`, you could write this:

```
<?php
$fruits = array (1 => "peach", "apple", "plum", "orange");
```

```
?>
```

Now peach is `1`, apple is `2`, plum is `3`, and orange is `4`. In addition to renumbering, the `=>` operator can be used to create string-indexed arrays. Sometimes your program will make more sense if you use string identifiers. For instance, an array of officers in an organization might use the position initials to identify each one:

```
<?php
$AcmeOfficers = array ("COB" => "Ralph Smith", "CEO" => "Carolyn
Jones", "CFO" =>
"Marilyn Kanter", "CIO" => "Hillman Tech", "VPO" => "George Ready");
echo $AcmeOfficers[CIO];
?>
```

In the example, the `echo` statement returns `Hillman Tech` because the string index `CIO` has been associated with the string literal `Hillman Tech` in the array. Note that the array index identifier has no quotation marks around it in the line with the `echo` statement.

Searching for Data with an Array

If you want to find a single element in an array, use a loop. When data are placed into an array using numbered elements, all you need is a loop and a counter, as the following example shows:

```
<?php
$parts=array("nuts" ,"bolts" ,"screws" ,"washers", "wingnuts");
while ($find != "screws") {
      $find = $parts[$counter];
      $counter += 1;
      }
echo $find;
?>
```

Looping through data from a database such as MySQL uses a similar process. If the data from the database is placed into an array, you will find it easy to get what you need using a loop. Often, you can use the same loop concept in a table to create a "built-in" array.

PHP Functions

PHP has built-in functions and user functions, just like JavaScript. Likewise, a number of other functions are built into PHP, and you can find them in the online manual at http://www.php.net. Building your own functions works like JavaScript user functions. For example, the following function brings a first and a last name together:

```
<?php
function fullName ($lastName,$firstName) {
      return $firstName . " " . $lastName;
}
echo fullName("Langley", "Harry");
?>
```

At this point, you know enough PHP that you can begin doing something with it in relationship to JavaScript. As you will see, most of the work done by JavaScript is limited, but you can effectively use JavaScript with forms to serve as a data-confirmation tool before sending the data to a PHP script.

Passing Data from JavaScript to PHP

In the role of a confirmation tool, JavaScript can check the content of forms before data is sent to a PHP script. Such a role is important for real-world forms because, if troublesome data are sent to a PHP script, the wrong data could end up in your database.

PHP recognizes data from an HTML form element using the same naming structure, but with the added dollar sign before the variable name. For example, if a form element is named this:

```
alpha
```

in an HTML page, it will be recognized as this in a PHP script:

```
$alpha
```

To see one role of JavaScript in gathering in information, these next two scripts, one an HTML page and the other a PHP script, show how data can originate in

JavaScript and be passed to PHP. First, the JavaScript function simply loads up a hidden form with a value. Note that the JavaScript variable `greet` is placed into a hidden form named `alpha`. The variable that is passed to the PHP script is `alpha` because PHP is getting its variable from a form named `alpha`. Thus, JavaScript makes a "bank-shot" to the form and then to PHP rather than directly to the PHP script.

getData.html

```
<html>
<head>
<title>Sending Data to PHP </title>
<script language="JavaScript">
function loadIt( ) {
      var greet="Hello from PHP.";
      document.storage.alpha.value=greet;
      }
</script>
</head>
<body onload="loadIt( )";>
<form name="storage" method=get action="greetings.php":>
      <input type=hidden name="alpha" >
      <input type=submit>
</form>
</body>
</html>
```

Save this script as getData.html. It uses the `get` method to open a PHP page named greetings.php. The PHP page immediately echoes whatever is in a variable named `$alpha`. As you will see, because `$alpha` is not declared or defined on the

PHP page, its content could be only what has been passed to it from the HTML page. Save the following page as greetings.php, and put it in the root directory or subdirectory in your root directory along with the getData.html page.

greetings.php

```
<html>
<body>
<?php
echo $alpha;
?>
</body>
</html>
```

When you run the getData.html script, press the Submit button to launch the PHP script. Your screen shows the line, "Hello from PHP."

Controlling Multiple PHP Pages with JavaScript

One use of JavaScript is to serve as a controller for PHP pages appearing in an HTML environment. Because PHP replaces a page that it is called from, one plan of action is to use JavaScript to create a number of buttons used to launch different PHP pages within a frameset. JavaScript can reside in the menu page, and PHP is called in the body page of a two-column frameset. At the same time, both the JavaScript page and the PHP page share a common external style sheet. Seven scripts are required for this project: an external CSS file, a frameset page, the menu page using JavaScript, a placeholder page, and three PHP pages demonstrating different features of PHP.

Cascading Style Sheet

Using a color scheme from *The Designer's Guide to Color Combinations* by Leslie Cabarga (North Light Books, 1999), the first step is to create a palette using the following colors. (All values are in hexadecimal.)

Color A	FFE699	Light tan
Color B	CC994C	Tan
Color C	CC0019	Deep red
Color D	00804C	Dark green
Color E	808080	Gray
Color F	000000	Black

This particular color scheme was selected because it contains black and gray, the colors of the buttons in HTML forms. The following style sheet incorporates the colors used in both the PHP and HTML pages.

controller.css

```
.bigRed {
      color: #cc0019;
      background-color: #ffe699;
      font-family: verdana;
      font-weight: bold;
      font-size: 18pt;
      }
```

```
.midRed {
color: #ffe699;
background-color: #cc0019;
font-family: verdana;
font-weight:bold;
font-size: 14pt;
}
```

Save the CSS style sheet in the same directory as the files for the HTML and PHP pages. All of the pages will employ the same style sheet.

This next HTML page establishes the frameset where the initial HTML pages—and eventually the PHP pages—will go.

controlSet.html

```
<html>
<frameset cols="25%,*" border=0>
      <frame name="menu" src="jMenu.html" frameborder=0 scrolling=0>
      <frame name="display" src="jsD.html" frameborder=0 scrolling=0>
</frameset>
</html>
```

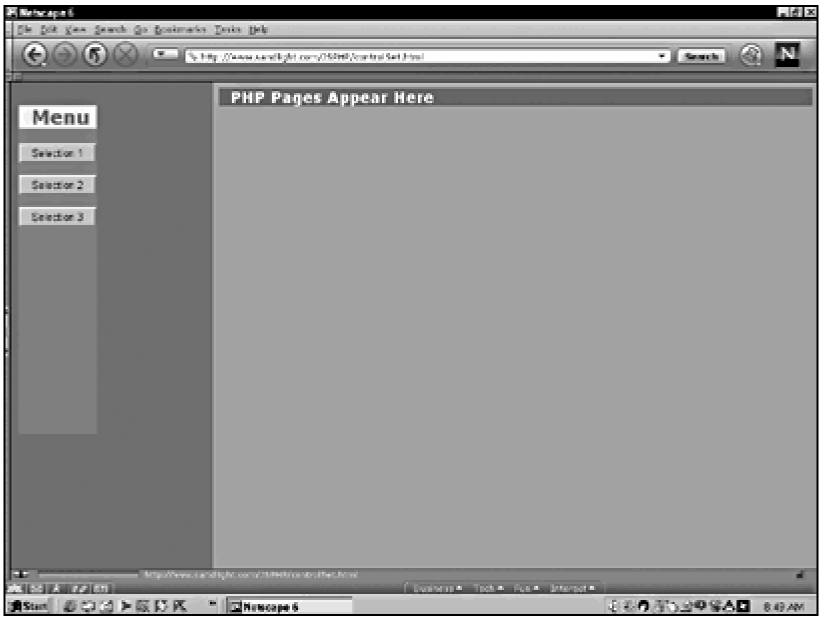
The left side of the frameset opens with a menu page and a blank HTML page in the right frame. All of the JavaScript controllers are in the menu. A simple function in JavaScript is used to launch any of the PHP pages in the right frame.

jMenu.html

```
<html>
<head>
<link rel="stylesheet" href="controller.css" type="text/css">
<script language="JavaScript">
      function loadPHP(url) {
      parent.display.location.href=url;
      }
</script>
</head>
<body bgcolor=#00804c>
<table border=0 height=70%>
      <tr align=left valign=top>
      <td bgcolor="#808080">
      <center>
            <div class="bigRed">&nbsp; Menu&nbsp;</div>
            </center>
            <br>
            <form><input type=button onClick="loadPHP('alpha.php')"
value="Selection 1">
            <p><input type=button onClick="loadPHP('beta.php')"
value="Selection 2">
            <p><input type=button onClick="loadPHP('gamma.php')"
value="Selection 3">
            </td>
      </tr>
</table>
</body>
</html>
```

The initial page in the right frame is a "dummy" page used to occupy space until one of the PHP pages is launched. Figure 14.2 shows how the initial page appears when the frameset is loaded.

Figure 14.2. The area on the right is reserved for PHP by an initial placeholder page.



jsD.html

```
<html>
<head> <title>Display Page</title>
<link rel="stylesheet" href="controller.css" type="text/css">
</head>
<body bgcolor=#cc994c>
      <div class=midRed> &nbsp PHP Pages Appear Here &nbsp </div>
</body>
</html>
```

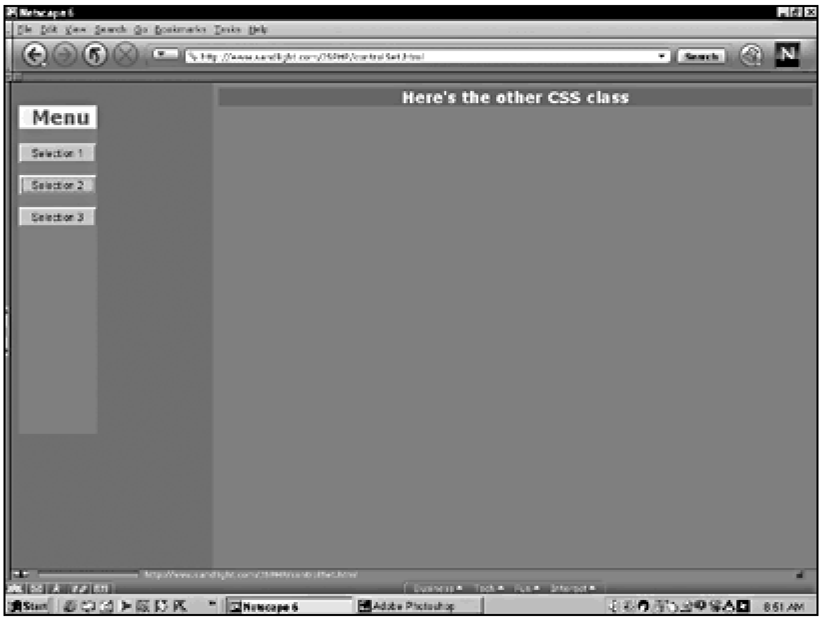
The first PHP page simply displays a message in plain text. However, it does load the same style sheet as the two previous HTML pages.

alpha.php

```
<html>
<head> <title>Control Menu</title>
<link rel="stylesheet" href="controller.css" type="text/css">
</head>
<body bgcolor=#808080>
<center>
<?php
      echo "This is plain text.";
?>
</center>
</body>
</html>
```

The second PHP page responds with a text message using one of the style sheet classes (see Figure 14.3).

Figure 14.3. A PHP page using the same CSS style sheet as the HTML page appears in the right column.



beta.php

```
<html>
<head> <title>Control Menu</title>
<link rel="stylesheet" href="controller.css" type="text/css">
</head>
<body bgcolor=#808080>
<center>
<?php
      echo "<div class=midRed>&nbsp Here's the other CSS class &nbsp
</div>";
?>
</center>
</body>
</html>
```

The third PHP page, like the second, responds with a message that indicates yet another of the CSS classes has been employed.

gamma.php

```
<html>
<head> <title>Control Menu</title>
<link rel="stylesheet" href="controller.css" type="text/css">
</head>
<body bgcolor=#808080>
<center>
<?php
    echo "<div class=bigRed>&nbsp CSS Text like the Menu &nbsp</div>";
?>
</center>
</body>
</html>
```

The role of JavaScript is to show how PHP pages can be controlled with a JavaScript function and how a common CSS file can provide a common style to front-end and back-end elements of a web site. In the next several sections, you will see how to check and send data from an HTML page to a PHP page using JavaScript for data verification.

JavaScript Form Preprocessing for PHP

Now that the concept of JavaScript's role with PHP is clear, it is time to do something more practical with JavaScript and PHP. This next set of scripts uses JavaScript as a preprocessor to make sure that a comment form being sent has all of the forms filled out and that the user remembers to include an "@" sign in her email address. The JavaScript also shows how to combine two functions into a single function that performs two tasks.

Before starting, you need to know a PHP function not yet discussed. The function `mail(e,s,m,h)` has four arguments:

- `e` Email address
- `s` Subject
- `m` Message
- `h` Header

Each of the arguments can be set as variables or strings in quotation marks. For example, this line would fire off an email to `joe@fuzz.com` with the other variables' contents going into the subject window, delivering a message, and creating a header:

```
mail("joe@fuzz.com",$subject,$talk,$whoMe);
```

To get started, I selected a French color scheme from Leslie Cabarga's international set of colors in *The Designer's Guide to Global Color Combinations* (North Light Books, 2001).

Color A	b3cfcc	Gray
Color B	6e9282	Gray-green
Color C	cc0019	Deep red
Color D	f2ede0	Cream
Color E	a44c3a	Terra cotta
Color F	000000	Black

In selecting the colors, I wanted to include a gray and black so that the color scheme would not clash with the form buttons.

mailer.css

```
.labelText {
font-family: verdana;
font-size:11pt;
font-weight:bold;
color:#6e9282;
}
.headerText {
font-family: verdana;
font-size:18pt;
color:#c0baae;
font-weight:bolder;
background-color:#a44c3a;
}
body {
background-color:#f2ede0;
}
```

When you have your style sheet created and stored out of the way, you are ready for the main JavaScript page. The following script checks what a user has placed into a form; if a problem is found, an alert message informs the user.

mailForm.html

```
<html>
<head>
<link rel="stylesheet" href="mailer.css" type="text/css">
<Title>Mail Form</title>
<script language="JavaScript">
//See if any of the document form elements are blank.
function checkBlank() {
      var flag=0;
      var count;
      for (count=0;count<3;count++) {
      if (document.mailme.elements[count].value=="") {
      flag=1;
            }
      }
            if (flag==1) {
            alert("Please fill all the forms.");
      }
}
//Make sure the @ is in the email address.
function checkAt() {
      var correct=0;
      var seek;
      var alpha=document.mailme.email.value;
      var ampFind=new String(alpha);
      var beta=ampFind.length;
      for (seek=0;seek<beta;seek++) {
            if(ampFind.charAt(seek)=="@") {
            var correct=1;
            }
      }
            if (correct==0) {
            alert("The @ is missing from your email address.");
      }
}
//Put the two functions together.
function checkAll() {
      checkBlank();
      checkAt();
}
</script>
</head>
<body>
<center>
<div class=headerText>
Mail Form
</div>
</center>
```

```
<div class=labelText>
<form name="mailme" method =get action="mailer.php">
Please enter your name:&nbsp; &nbsp
<input type="text" name="name" size=24><p>
Please enter your email address: &nbsp; &nbsp
<input type ="text" name="email" size=40><p>
We'd like to hear what you have to say: <br>
<textarea name="comments" rows=10 cols=80></textarea><P>
<input type="button" value="Click to Verify:" onClick="checkAll()">
<input type="submit">
</form>
</div>
</body>
</html>
```

What's Going On?

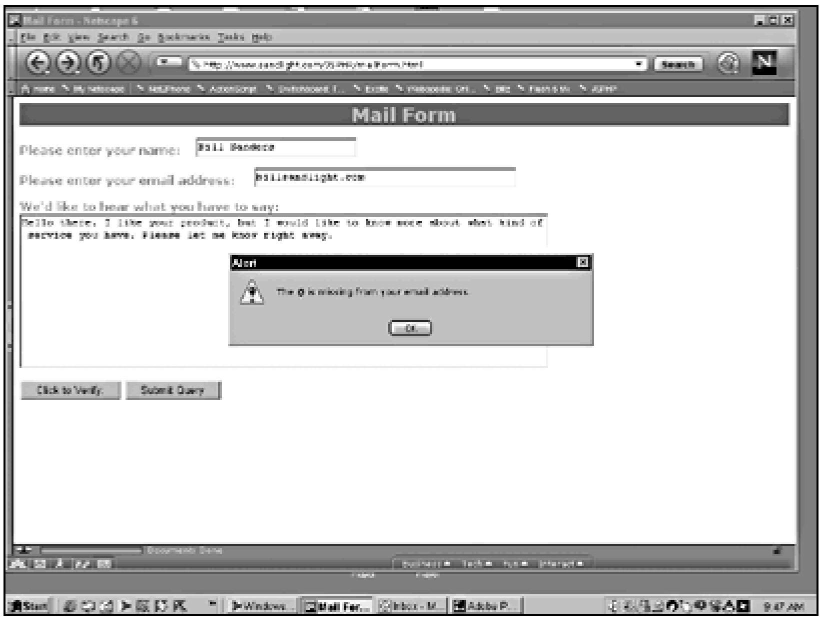
The script is a bit long, but it is really quite simple. The first function, `checkBlank( )` sets up a loop that examines the first three elements of the `mailme` form. (The other elements of the form are buttons and have no input data.) `checkBlank( )` looks for the empty quotations (`""`), and if it finds any, it sets a flag variable named `flag`. The second function, `checkAt( )` examines what the user wrote into the email window to make sure that the `@` is in place. To do this, the script first placed the contents of the form into a variable named `alpha`:

```
var alpha=document.mailme.email.value;
```

That variable is then made into a string object in the variable `ampFind` so that all of the string functions can be used to find the length of the email address and check each character to see if it contains an @ character. If it does, a flag variable named `correct` is set. In this case, the alert message is sent only if the `flag` variable is not set.

The last bit of JavaScript combines both forms so that a single click will check for both types of errors. If an error is detected, a warning appears, as shown in Figure 14.4.

Figure 14.4. The JavaScript detects a mistake and sends an alert message to the viewer.


Mail Form

Please enter your name: Bill Sanders

Please enter your email address: billsandlight.com

We'd like to hear what you have to say:

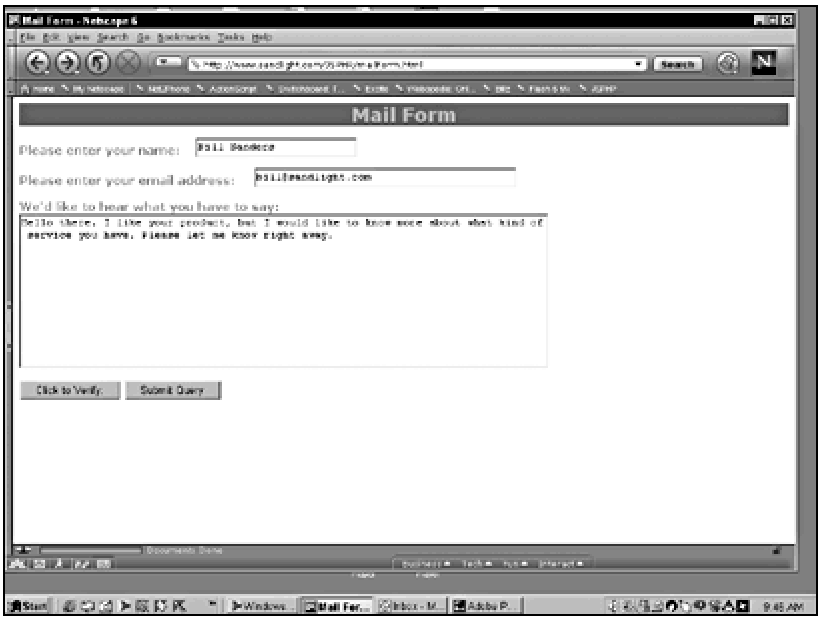
Hello there, I like your product, but I would like to know more about what kind of service you have. Please let me know right away.

Alert

The @ is missing from your email address.

When the viewer sees the error message, she has an opportunity to correct it before submitting the data to a PHP script. Figure 14.5 shows how the corrected page appears before submitting it.

Figure 14.5. When the problem is solved, no error messages appear on the screen and the user may press the Submit Query button.


Mail Form

Please enter your name: Bill Sanders

Please enter your email address: bill@sandlight.com

We'd like to hear what you have to say:

Hello there, I like your product, but I would like to know more about what kind of service you have. Please let me know right away.

mailer.php

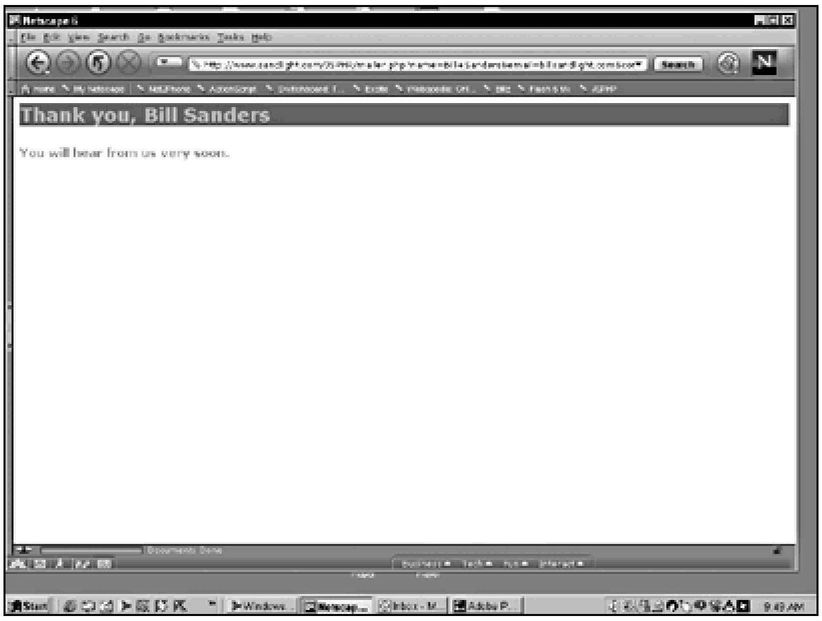
```
<html>
<head>
<link rel="stylesheet" href="mailer.css" type="text/css">
</head>
<body>
<?php
$recipient=$email;
$reply="Dear " . $name . ", \n\n";
```

```
$reply .= "Your information has been received. Thank you for filling
out the form.";
$reply .= " We will have someone go over comments ";
$reply .= "and get back to you by email soon.";
$reply .= "\n\nSincerely, \nSuzy Q. Less \nPublic Relations";
mail($recipient, "Your Comments",$reply);
$bundle=$comments . "\n\n": . $name . "\n\n" . $email;
mail("yourCompany@email.com","Customer",$bundle);
?>
<p class=headerText>
<?php echo " Thank you, " .$name; ?>
</p>
<p class=labelText>
<?php echo "You will hear from us very soon."; ?>
</p>
</body>
</html>
```

What's Going On?

All three of the data-entry form elements in the HTML page are passed to the PHP page. The form elements `name`, `email`, and `comments` are received by PHP as `$name`, `$email`, and `$comments`. These are then used in the PHP script to form a reply, address an email, and pass information to the owner of the web site. Hence, you will see *two* instances of the `mail( )` function. The first sends an email to the person who filled out the forms, and the second sends one to the owner of the web site. Figure 14.6 shows the immediate response that the user sees when he submits the form.

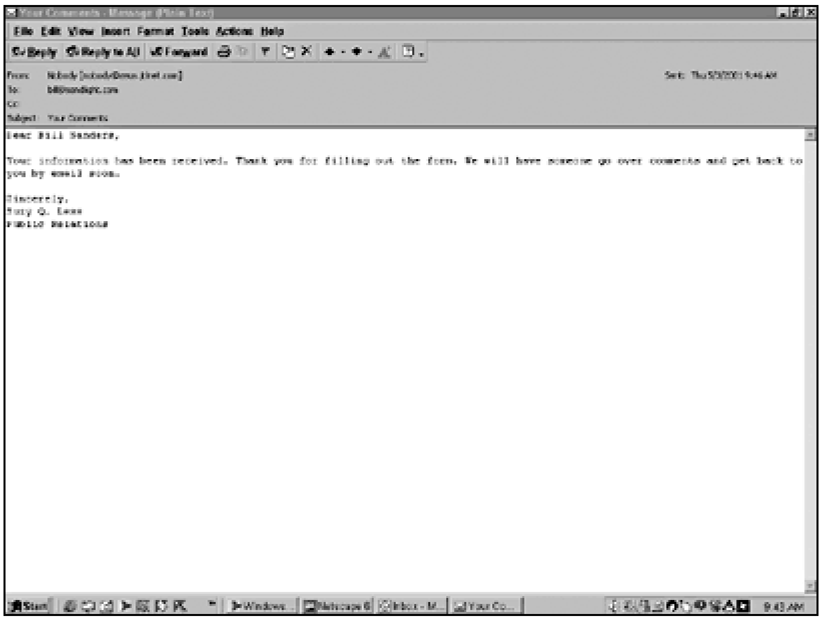
Figure 14.6. A reply on the screen indicates that the information has been sent.


Thank you, Bill Sanders

You will hear from us very soon.

Also note how the PHP code is interspersed with the HTML tags. It is extremely important to remember to put in the semicolons after each PHP line, where appropriate. The same information that is used in the email reply is used to place a message on the screen as soon as the data is submitted. The person who clicks the Submit Query button not only gets a reply immediately on the browser that he's viewing, as shown in Figure 14.6, but he also gets an immediate email signed by a public relations representative, shown in Figure 14.7.

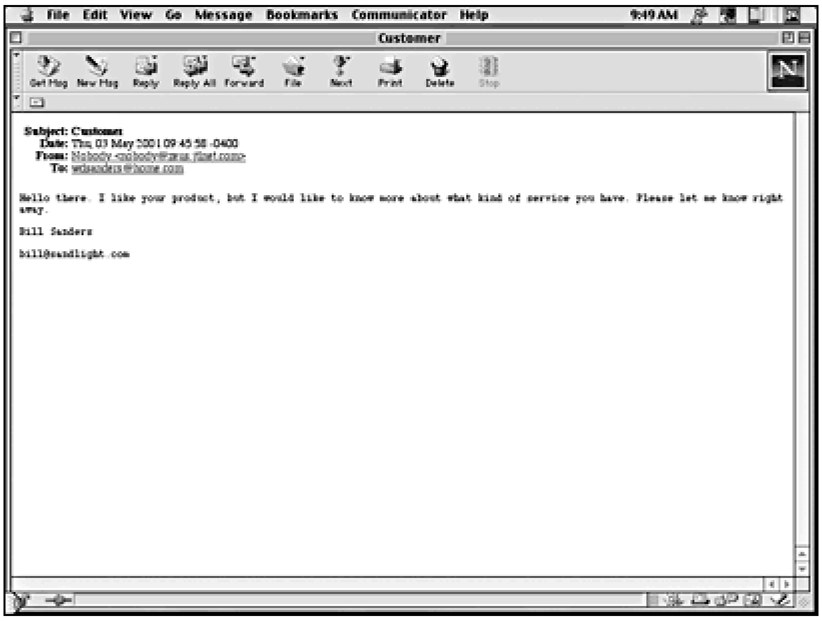
Figure 14.7. An immediate email is sent to the viewer.



```
From:    Nobody [nobody@mus.jlnet.com]          Sent: Thu 5/3/2001 9:46 AM
To:      bill@sandlight.com
Cc:
Subject: Your Comments

Dear Bill Sanders,

Your information has been received. Thank you for filling out the form. We will have someone go over comments and get back to
you by email soon.

Sincerely,
Suzy Q. Less
Public Relations
```

The web site owner gets all of the information sent by the viewer. It's a barebones letter with the name, email, and comment variables displayed as a message, as shown in Figure 14.8.

Figure 14.8. The email with the viewer's information is sent to an email address determined by the site owner.



```
Subject: Customer
   Date: Thu, 03 May 2001 09:45:58 -0400
   From: Nobody <nobody@mus.jlnet.com>
     To: wdsanders@home.com

Hello there. I like your product, but I would like to know more about what kind of service you have. Please let me know right
away.

Bill Sanders

bill@sandlight.com
```

JavaScript, PHP, and MySQL

With JavaScript's capability to check data before it is sent to a PHP script, it serves a valuable role in making sure that data sent from an HTML page are the data that you want in a database. When the data are in the database, you can recall them using HTML aided by JavaScript as a front end.

An open source relational database-management system (RDBMS) typically used with PHP is MySQL. MySQL can be downloaded and used for professional database development. While you can use it without charge for learning and most personal uses, a small fee is required for commercial uses. The current version as I write this page is 3.23.37, and you can be assured that it will be upgraded by the time you read this passage. Check http://www.mysql.com for the latest version.

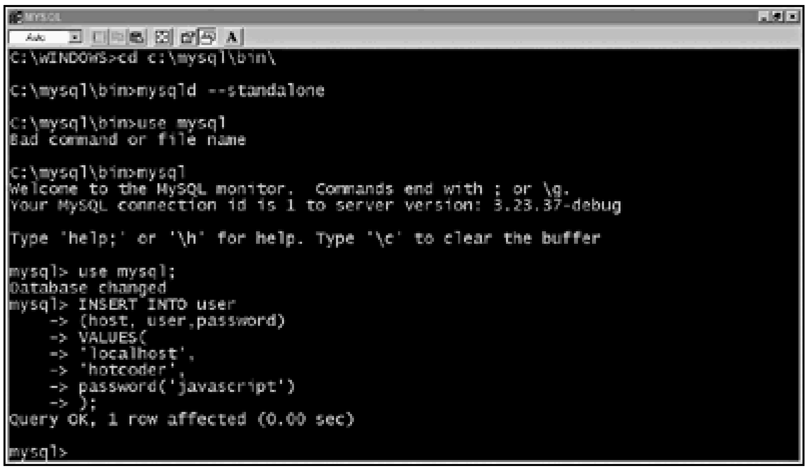
Fundamental MySQL Commands

One way to learn and practice the basic MySQL commands is to put the MySQL sever on your computer. Download the version of MySQL for your operating system from http://www.mysql.com, install it following the installation prompts, and you should be all set to start creating databases and tables in MySQL.

Use the following steps if you're running everything on your Windows PC for the initial setup:

1. Select Start, Programs, Accessories, MS-DOS Prompt to open a DOS window.
2. In the DOS window, type `cd c:\mysql\bin\` and press Enter. You will see the DOS window with white type and a black background.
3. Type `mysqld --standalone` and press Enter. (Be sure that you have a space between `mysqld` and `--`).
4. Enter the MySQL monitor by typing `mysql` and pressing Enter. If you've successfully entered the monitor, you will see a `mysql>` prompt. All commands in the MySQL monitor must be terminated with a semicolon.
5. (See Figure 14.9.)

Figure 14.9. From the MySQL monitor, you can enter MySQL commands.



```
C:\WINDOWS>cd c:\mysql\bin\

C:\mysql\bin>mysqld --standalone

C:\mysql\bin>use mysql
Bad command or file name

C:\mysql\bin>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 3.23.37-debug

Type 'help;' or '\h' for help. Type '\c' to clear the buffer

mysql> use mysql;
Database changed
mysql> INSERT INTO user
    -> (host, user,password)
    -> VALUES(
    -> 'localhost',
    -> 'hotcoder',
    -> password('javascript')
    -> );
Query OK, 1 row affected (0.00 sec)

mysql>
```

6. This next step creates a new database user, and you need to do it just right. Look at Figure 14.9, and type in the same code, pressing the Enter key at the end of each line.

7. Once you have inserted the user, exit the MySQL monitor by typing `exit`. You should see the `C:\mysql\bin>` cursor. Now just type in `mysql` and press Enter.

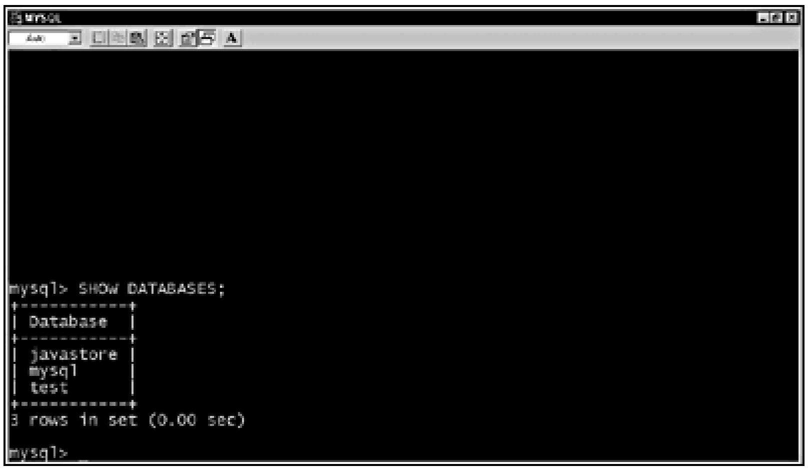
Making Your Own Databases and Tables

When you are in the MySQL monitor, you should be able to create databases and tables and add records to the tables. To create a database, type in the following:

```
mysql> CREATE DATABASE javastore;
```

The name of the database just created is javastore, and it is seen by issuing a `SHOW DATABASES` command (see Figure 14.10).

Figure 14.10. All of the databases in MySQL are displayed, including the one just created.



```
mysql> SHOW DATABASES;
+-----------+
| Database  |
+-----------+
| javastore |
| mysql     |
| test      |
+-----------+
3 rows in set (0.00 sec)

mysql> _
```

The command displays all the databases established. The databases named mysql and test are established by default. The database javastore is the one just created. Creating new databases with MySQL is quite simple.

To add a table to the database, first issue a `USE javastore`; command to select the desired database. Next you will create a table that contains all the fields that you will need for a name and address database. You will need the following nine fields:

- Identity number
- Last name
- First name
- Address
- City
- State
- ZIP code
- Phone
- Email

The purpose is to create a table to show the email addresses of a list of people stored in the database. Each field must include a name, a data type, and the length of the field for fixed length types. Table 14.2 lists a sample of some of the data types in MySQL.

Table 14.2. MySQL Column (Field) Types of Data

Data Type	Description
`INT(n)`	Integer number
`FLOAT`	Floating point (single precision)
`DOUBLE`	Floating point (double precision)
`DECIMAL(n,dp)`	Float stored as string
`CHAR(n)`	Fixed-length string from 0 to 255
`VARCHAR(n)`	Variable-length string from 0 to 255
`TEXT`	Text field from 0 to 65535 bytes

Create a table using the following format:

```
mysql> CREATE TABLE tabldogname (field1 DATATYPE(N),
 field2 DATATYPE(N)
, etc.);
```

Because the table used in this example has nine fields, you will need nine unique field names. Other than the first field that will be an integer number (`INT`) and the state and ZIP code fields that will be fixed-length strings (`CHAR`), the values will all be variable-length strings (`VARCHAR`).

To create the table and all of its fields, be sure that javastore (`USE javastore`) is the selected database, and enter the following in the MySQL monitor:

```
mysql> CREATE TABLE clientlist (
    -> id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    -> lname VARCHAR(20),
    -> fname VARCHAR(20),
    -> address VARCHAR(40),
    -> city VARCHAR(30),
   -> state CHAR(2),
   -> zip CHAR(5),
   -> phone VARCHAR(15),
   -> email VARCHAR(35)
   -> );
```

After you press the Enter key after the last semicolon, you should see the following:

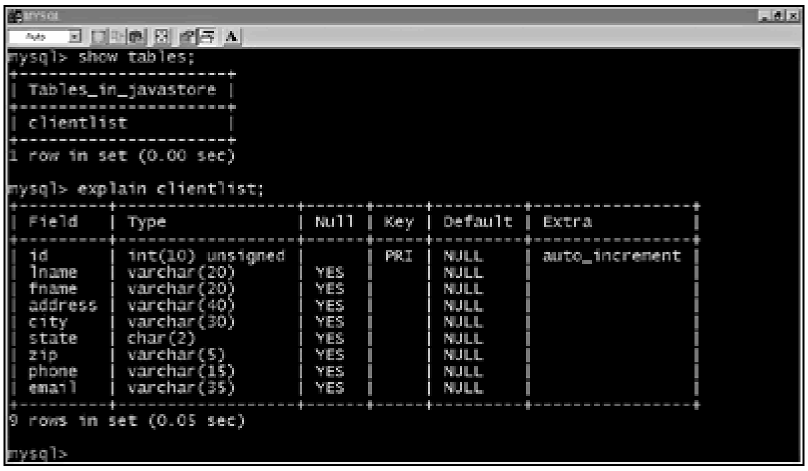
```
   Query OK, 0 rows affected (0.00 sec)
   mysql>
```

If you get any kind of error, your table will not be created. To check whether your table is actually formed, type in the command **`SHOW TABLES`**. The following shows the sequence:

```
mysql> SHOW TABLES;
+-------------------------+
| Tables_in_javastore     |
+-------------------------+
| clientlist              |
+-------------------------+
1 row in set (0.00 sec)
```

To check all the fields in your table, enter the command **`EXPLAIN clientlist`** and press Enter. Figure 14.11 shows what you will see if you have entered your data correctly.

Figure 14.11. All of the fields in the database are displayed.



```
mysql> show tables;
+---------------------+
| Tables_in_javastore |
+---------------------+
| clientlist          |
+---------------------+
1 row in set (0.00 sec)

mysql> explain clientlist;
+---------+------------------+------+-----+---------+----------------+
| Field   | Type             | Null | Key | Default | Extra          |
+---------+------------------+------+-----+---------+----------------+
| id      | int(10) unsigned |      | PRI | NULL    | auto_increment |
| lname   | varchar(20)      | YES  |     | NULL    |                |
| fname   | varchar(20)      | YES  |     | NULL    |                |
| address | varchar(40)      | YES  |     | NULL    |                |
| city    | varchar(30)      | YES  |     | NULL    |                |
| state   | char(2)          | YES  |     | NULL    |                |
| zip     | varchar(5)       | YES  |     | NULL    |                |
| phone   | varchar(15)      | YES  |     | NULL    |                |
| email   | varchar(35)      | YES  |     | NULL    |                |
+---------+------------------+------+-----+---------+----------------+
9 rows in set (0.05 sec)

mysql>
```

Entering and Retrieving Records

After you have created a database and table, to make it useful you need a way to enter and retrieve records. To insert records into a table, first be sure that the correct database is selected and then use the following format to enter records:

```
mysql> INSERT INTO tabldogname VALUES('field1', 'field2');
```

The data that makes up `field1` and `field2` originates in the HTML page, is checked with JavaScript, and then is sent to a PHP file. So, besides entering the data as literals, variables passed from JavaScript/HTML to PHP to MySQL can be used for entering data into MySQL as well. Moreover, you will find doing so much easier than using the MySQL monitor. The following code shows the correct procedure for entering a record and the feedback from the MySQL monitor:

```
mysql> INSERT INTO clientlist VALUES(
    -> null,
    -> 'Smith',
    -> 'Joe',
    -> '123 Maple Street',
    -> 'Taft',
    -> 'CA',
    -> '92012',
    -> '619-555-4502',
    -> 'josm@hubahuba.com'
    -> );
Query OK, 1 row affected (0.05 sec)

mysql (prompt on screen)
```

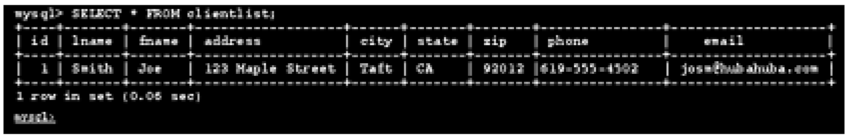
The important value entry is the `null` in the first field. No matter what record you are entering, when you put in `null` in an `AUTO_INCREMENT` field, the value for that record is increased by 1. This automatic value in a primary key means that you don't have to keep track of the unique value that each record will have. So, if you have two people named Joe Smith, each will have a unique number in his `id` field to distinguish him. Also, when using more than one table in a database, the `id` field will keep your data in the proper relationship. So, if you have all the names in one table and other information about the same person in another, the common `id` field with `AUTO_INCREMENT` will help keep your records straight.

To see the record that you put in, you can specify different characteristics of the record. To see all of the fields and all of the records, use the asterisk (*) as a wildcard character. For example, to see all of the records in your table, you would type this:

```
SELECT * FROM clientlist;
```

Figure 14.12 shows what you would see given the information entered in the example.

Figure 14.12. All of the fields and their values are displayed using the wildcard character.



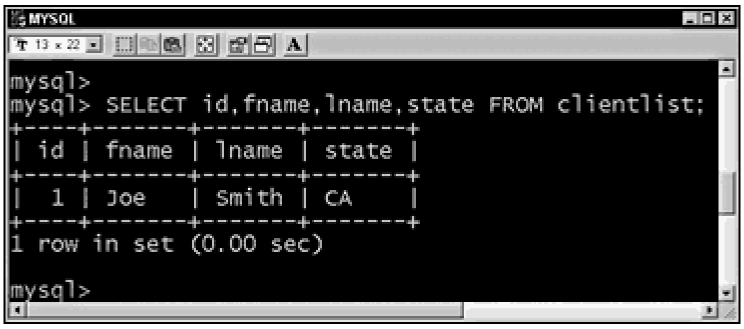
```
mysql> SELECT * FROM clientlist;
+----+-------+-------+-------------------+-------+-------+-------+--------------+-------------------+
| id | lname | fname | address           | city  | state | zip   | phone        |      email        |
+----+-------+-------+-------------------+-------+-------+-------+--------------+-------------------+
|  1 | Smith | Joe   | 123 Maple Street  | Taft  | CA    | 92012 |619-555-4502  | josm@hubahuba.com |
+----+-------+-------+-------------------+-------+-------+-------+--------------+-------------------+
1 row in set (0.05 sec)
mysql>
```

You can also select just portions of the table in any order your want. For example, you could enter this, and you would get only the selected fields in the order that you listed them:

```
SELECT id,fname,lname,state FROM clientlist;
```

Figure 14.13 shows the outcome from the previous command.

Figure 14.13. Only the selected fields are displayed in the order specified in the command.



```
mysql>
mysql> SELECT id,fname,lname,state FROM clientlist;
+----+-------+-------+-------+
| id | fname | lname | state |
+----+-------+-------+-------+
|  1 | Joe   | Smith | CA    |
+----+-------+-------+-------+
1 row in set (0.00 sec)

mysql>
```

A third SELECTION command allows you to select both a field and a field value. For example, if you wanted to find a phone number in which the person's first name was Joe, you could enter this:

```
SELECT phone FROM clientlist WHERE fname="Joe";
```

Experimenting with different MySQL commands helps you understand the PHP commands that put data into and pull data out of a MySQL database. The data originates in an HTML page and is checked by a JavaScript verification routine. Next, the HTML data is sent to a PHP page, where the variables are used to put information into the database or take it out.

PHP and MySQL

PHP contains a number of functions designed to communicate with a MySQL database. A full list of PHP commands associated with MySQL can be found at http://www.php.net/manual/ref.mysql.php, but for the purposes of this introduction, you will need only a few of the PHP-MySQL functions. Table 14.3 lists the ones used in this chapter.

Table 14.3. PHP MySQL Functions

Function	Result
mysql_connect	Establishes a connection to a MySQL server
mysql_query	Send a MySQL query
mysql_result	Gets result data
mysql_num_rows	Returns the number of rows in result
mysql_select_db	Chooses a MySQL database

PHP Connection to MySQL

The first MySQL function in PHP to learn is mysql_connect(). As you can guess from the name, the function establishes a connection between PHP and the database. To make the connection, the mysql_connect() function requires three arguments: server, user, and password. By placing the names of the server, user, and password into variables, it is easier to make sure that you have the names right and also to change the server, user, or password.

PHP uses the die() function to find out whether a connection is made. So, the or die (message) function is often used to help locate connection problems to the different links that must be made to MySQL. Use the following PHP script to test your connection:

```
<HTML>
<head>
<Title>Test MySQL Connection</title>
</head>
<body bgcolor="orangered">
<center>
<h1>
<?php
```

```
$server="localhost";
$user="hotcoder";
$pass="javascript";
$connectMy=mysql_connect($server,$user,$pass) or die("No connection
made.");
if ($connectMy) {
$reply="Connection confirmed.";
}
echo "$reply";
?>
</h1>
</center>
</body>
</html>
```

Save the script in the root directory of your server, or a directory within the root directory, as connectTest.php. I added a subdirectory to my root directory to keep everything clear:

```
http://localhost/jsphp/connectTest.php
```

The server name localhost is one that I used on my computer serving as host and client. On a professional hosting service, you would use your domain name as the root directory and whatever subdirectory you use to store your scripts for a given project. In the previous examples using MySQL, I used the database name of javastore and the username of hotcoder with the password javascript. You can use any names that you want as long as you establish them in the MySQL database first. With the connectTest.php script, you should see a big "Connection confirmed" in the middle of your screen if you have established the MySQL database message.

If your connection fails, check the following:

- Be sure that you have entered your host (server), user, and passwords into MySQL as shown in Figure 14.8, or in your hosting service's MySQL application for setting usernames and passwords. (The hostname is established by the hosting service.)
- Check to be sure that you have placed the PHP file in the root directory. If the PHP file is in the wrong place, it won't display at all. This has nothing to do with a failed connection to MySQL.
- Make sure that your Apache server and MySQL monitor are running if you are using your computer as a client and server.

Selecting the Database with PHP

Use the PHP function `mysql_select_db(databasename, connection)` to select the that database you want to use. The database name is the name that you have given the database on the host. The connection name is the variable where the `mysql_connect()` data is stored. This example suppresses the automatic feedback to the web page using an "at" sign (@) in front of the database select function. Only the `die()` function message is displayed with the @ symbol at the beginning of the select database function.

```
<?php
```

```
//Define elements
$server="localhost";              //Host name
$user="hotcoder";          //User name
$pass="javascript";                  //Password
$javastore ="javastore";    //database name

//Connect to MySQL
$connectMy = mysql_connect($server, $user, $pass);

//Choose database
$dataB=@mysql_select_db($javastore,$connectMy) or die ("Database not
responding");

if ($dataB) {
$reply="Database is selected and ready to go.";
}
echo "$reply";
?>
```

When you can connect to MySQL and the desired database, the rest is pretty smooth sailing. You will find that far fewer problems occur when using a hosting service than when attempting to run both the Apache server and MySQL monitor on your own computer.

Inserting Tables into the Database

The same MySQL commands used in the previous section on creating a table now can be used in a PHP script to create a table for the database. The two-step process involves first creating a string with the MySQL commands. Usually, we prefer to use a variable for the table name so that when a different table needs to be inserted, all we need to do is to substitute table names. In the first step, the string with the command elements replicates the same statements and arguments that would be done in the MySQL monitor. Other than the variable name substituted for the actual table name, the script between the quotation marks is exactly what would be written in the MySQL monitor when creating a table. Next, use the `mysql_query(query,connect)` function to send the command to MySQL. The following script creates a table named dognames in the database named javastore. Substitute your own names for the user, password, database, and table, if you want. If you are using your own hosting service, be sure to use the root names that your hosting service assigns.

```
<HTML>
<head>
<Title>Make Table</title>
</head>
<body bgcolor="orangered">
<center>
<h1>
<?php
//Define elements
$server="localhost";          //Host name
$user="hotcoder";             //User name
$pass="javascript";                     //Password
$useBase ="javastore";      //database name
$tabName="dognames";     //table name
//Connect to MySQL
$connectMy = mysql_connect($server, $user, $pass);
```

```
//Choose database
$dataB=mysql_select_db($useBase,$connectMy);
// MySQL command in variable
$sql = "CREATE TABLE $tabName (dogname varchar(30), breed
varchar(20))";
// Effect Command
$result = @mysql_query($sql,$connectMy) or die("Table not created.");
if ($result) {
echo "Table has been created.";
    }
?>
</h1>
</center>
</body>
</html>
```

Save the script as makeTable.php, and launch it from your browser. If you get any result other than "Table has been created," check your script and the names of the different connections.

Inserting Records into a Table Using PHP and JavaScript

To add records, you will need data sent from HTML to a PHP script. As you have seen in previous sections, you can pass data between HTML and PHP by using the `GET` method in a form. The role of JavaScript is to verify the data. Both the PHP script and the HTML page use the following color palette saved in an external CSS file:

Color A	FFB3E6	Pink light tan
Color B	260000	Dark, dark red
Color C	A65900	Tan
Color D	664066	Gray
Color E	000000	Black

First, create your CSS page and save it in the root directory.

adder.css

```
h1 {
font-size:18pt;
font-family:verdana;
color:#a65900;
background-color:#260000;
font-weight:bold
}

.bodText {
font-family: verdana;
font-size:11pt;
font-weight:bold;
color:#ffb3e6;
background-color:black
}

body {
background-color: #a65900;
}
```

Next, create your HTML page with JavaScript providing verification services, but with the main work being done by HTML forms.

recordAdder.html

```
<HTML>
<head>
<link rel="stylesheet" href="adder.css" type="text/css">
<Title>Record Adder</title>
<script language="JavaScript">
function checkIt() {
for(var x=0;x<2;x++) {
var checker=document.forms[0].elements[x].value;
if (checker=="") {
alert("Please fill in all windows.");
              }
       }
}
</script>
</head>
<body>
<center>
<h1>&nbsp Add Records &nbsp</h1>
<form name="records" method=get action="newRecord.php">
<span class=bodText>&nbsp Dog's Name: &nbsp</span>
<input type=text name="name">
<br>
<span class=bodText>&nbsp Dog's Breed:&nbsp</span>
<input type=text name="breed"><p>
<input type="button" value="Verify" onClick="checkIt()">
<input type=submit>
</form>
</body>
</html>
```

The PHP script gets two variables from HTML. One is `name` and the other is `breed`, stored in the form element names. The two HTML variables are transformed into `$name` and `$breed` in PHP. The `$name` variable is part of a record that goes into the field `dogname;` `$breed` goes into the field `breed` in a table named dognames. The general MySQL statements are loaded into a variable, `$sql`. Then, using the `mysql_query` function, send the data into MySQL as a record.

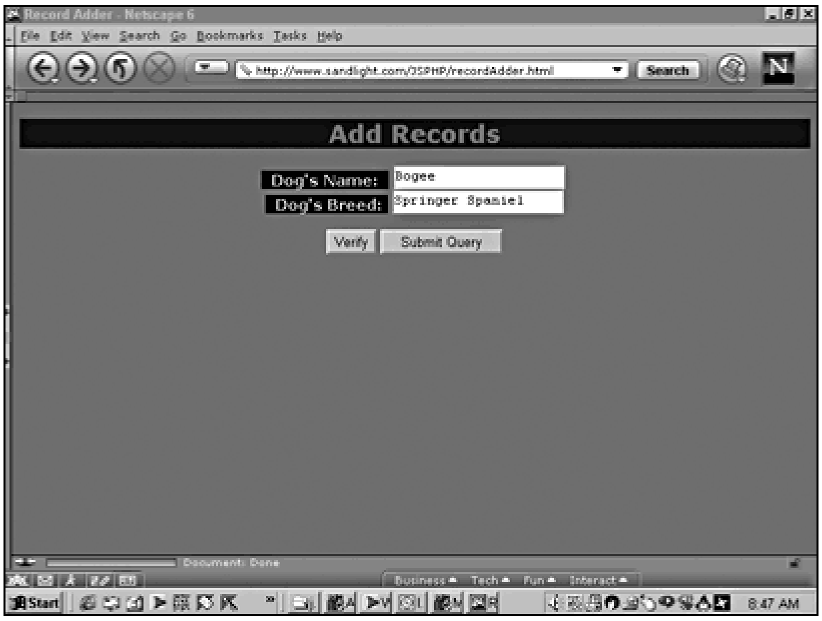
addRecord.php

```
<html>
<head>
<link rel="stylesheet" href="adder.css" type="text/css">
<Title>Add new record</title>
</head>
<body>
<?php
//Define elements
$server="localhost";             //Host name
$user="hotcoder";          //User name
$pass="javascript";                  //Password
$dataB ="javastore";    //database name
$ctable="dognames";         //table name
//Connect to MySQL
$connectMy = mysql_connect($server, $user, $pass);
//Choose database
```

```
mysql_select_db($dataB,$connectMy);
//Variables from JavaScript become data for MySQL.
$sql="INSERT INTO $ctable (dogname, breed) VALUES('$name','$breed')";
//Use the query function to send record to MySQL.
mysql_query($sql,$connectMy);
$msg=$name . " has been added.";
echo "<span class=bodText>$msg</span>";
?>
</body>
</html>
```

Save the PHP file as newRecord.php, and put it in your root directory or a folder in the root directory along with the CSS and HTML pages. Figure 14.14 shows the initial screen. As soon as the Submit Query button is clicked, the data is sent into the database. With more entries by the user, the role of JavaScript verification becomes more important.

Figure 14.14. The Verify button is JavaScript's contribution to the page.



Selecting Records from a Table

Getting records from a MySQL database via PHP and sent to the browser works very much like the routine for recording records, but the order is reversed. The PHP script first makes the connection and selects the database. Then the script queries the MySQL table using the `SELECT` statement. To get a better understanding of what happens when selecting data from a table, everything on the table is selected using the wildcard asterisk (*) and is stored in a variable. After getting the data, the next set of statements in PHP gets the precise data that you need.

Initially, `SELECT * FROM TABLE dognames` pulls out all the information in the table and stores it in a variable named `$result`. Next, the `mysql_result()` function returns the data in one cell from a result set. The result set is broken down into the row and field. For example, this statement stores the returns of the contents of row 5 (the sixth row because the rows begin with row 0) of the field titled `dogname` from the result set of the table named dognames stored in `$result`.

```
$name=(mysql_result($result,5,"dogname"));
```

You can change the row value (`5`) to anything that you want, as long as your number is from 0 to the number of rows in your table minus 1.

Because the field `dogname` contains a pooch's name, the variable `$name` now should contain the name of the dog that you put into the sixth row (row 5) of the table. The PHP script passes the name back to the browser for viewing.

Passing variables from HTML to PHP was demonstrated in the last example, and this next set of scripts uses the same technique. Thus, to get the results from a row in a table in a database, you have to send only a single number from HTML to PHP. To get started, the following color palette has been selected:

Color A	004CB3	Blue
Color B	FFFFE1	Pale, pale yellow
Color C	A6B380	Muddy green
Color D	D95900	Dark tan
Color E	C5FFC0	Pale blue/green
Color F	000000	Black

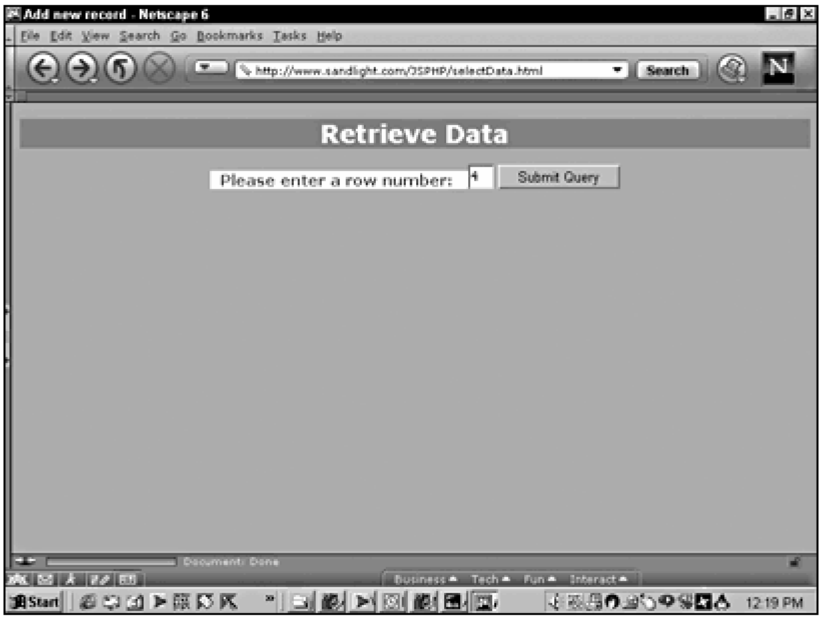
First, using colors from the palette, create the following CSS file, and save it and the other two files in the root directory or a subdirectory within your root directory.

select.css

```
.bodyText {
font-family:verdana;
color:#004cb3;
background-color: #ffffe1;
font-size:11pt;
font-weight:bold
}

h1 {
font-family:verdana;
font-size:18pt;
color:#c5ffc0;
background-color:#d95900
}
```

Next, you need an HTML page to pass the data from the user to the PHP page. As noted previously, it requires nothing but a number to be passed to find a row. Like in the previous HTML script, the data value is passed through the name of a form element. (See Figure 14.15.) Look for the `dn` name in the text form element.

Figure 14.15. The number entered in the text window will be sent to the PHP page as a variable.


Add new record - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://www.sandlight.com/JSPHP/selectData.html
Search
Retrieve Data
Please enter a row number:
4
Submit Query
Document: Done
12:19 PM

selectData.html

```
<html>
<head>
<link rel="stylesheet" href="select.css" type="text/css">
<Title>Add new record</title>
</head>
<body bgcolor=#a6b380>
<center>
<h1>&nbsp Retrieve Data &nbsp</h1>
<form name="eye" method=GET action="getData.php">
<span class=bodyText>&nbsp Please enter a row number: &nbsp </span>
<input type="text" size=2 name="dn">
<input type=submit>
</form>
</body>
</html>
```

The key in the HTML file, of course, is the form element with the `dn` variable name that will be used by PHP to find the desired row. Finally, the PHP page will take the `dn` variable and display the row information shown in the following script.

getData.php

```
<html>
<head>
<title>Get the Data</title>
<link rel="stylesheet" href="select.css" type="text/css">
</head>
<body bgcolor=#d95900>
<?php
$server="localhost";                          //Host name
$user="hotcoder";            //User name
$pass="javascript";                           //Password
$dataB ="javastore";         //database name
$ctable="dognames";                         //table name
//Connect to MySQL
$connectMy = mysql_connect($server, $user, $pass);
//Select database
mysql_select_db($dataB,$connectMy);
```

```
//Query specific table in database.
$result = mysql_query("SELECT * FROM dognames",$connectMy);
//Get method sends variable data 'dn' from HTML to PHP script—$dn.
$name=(mysql_result($result,$dn,"dogname"));
//Form variable value remains constant for both fields.
$breed  =(mysql_result($result,$dn ,"breed"));
$msg=$name . " is a " . $breed;
echo "<span class=bodyText>$msg</span>";
?>
</body>
</html>
```

Summary

JavaScript's role in working with server-side languages such as PHP is one of preprocessing. Before data is passed to PHP to be placed into a database, the data needs to be examined to find out whether the user entered the text and values correctly. In this way, when the data is passed to PHP and on to a MySQL database, the user is not surprised to find either empty fields or mistakes that require re-entering data.

However, while JavaScript does have a pivotal role in communications with the server-side elements in a script, the data is passed by the form's submit routine. Unfortunately, the `submit()` function in JavaScript will not set in motion the necessary elements to send data in the HTML page's forms to the PHP script to be processed. However, you can use the `submit()` function to launch a PHP script where passing variables in not a requirement.

This very short introduction to PHP and JavaScript is meant to illuminate the role of JavaScript as a preprocessor and to show you how to use the basic elements of PHP. If you are interested in doing more with PHP and JavaScript, many excellent books are available, and several sites are dedicated to PHP where you can learn much more.

Chapter 15. Using ASP with JavaScript

CONTENTS>>

In this book, JavaScript has been examined as a client-side scripting language, and in this chapter it will continue to be so. As with PHP, discussed in Chapter 14, "Using PHP with JavaScript," the role of JavaScript is that of a preprocessor in dealing with Active Server Pages (ASP). Before data are sent to a back-end or server-side script, you want to make sure that the data entered are what you want. If your data are examined by a server-side script, a good deal of bandwidth is wasted as messages are passed back and forth between the client's browser and the server. However, if JavaScript makes sure that the data are clean, then all of the preprocessing is done in the client's browser, with nothing wasted in between.

ASP works with Microsoft NT Servers. ASP pages are usually associated with a language called VBScript, but you can find JavaScript or XML in ASP pages. Like the PHP pages discussed in the previous chapter, ASP pages are saved in the web server's root directory, but with the .asp extension instead of .php. Depending on the server you're using, your root directory will vary. The domain that I use for Active Server Pages is `active1.hartford.edu`, and I keep my pages in a folder named hotjava. If my ASP page is saved as whatzUP.asp, to access it I would enter this:

```
http://active1.hartford.edu/hotjava/whatzUP.asp
```

The addressing looks a lot like what you use for HTML, except that the extensions end with .asp instead of .html.

NOTE

Before you get started, you will need an NT server. On some versions of Windows (such as Windows 2000), you can run Windows NT software, but on others you cannot. To save yourself some grief, I recommend signing up for a professional Windows NT hosting service. Not only will you get real-world experience using ASP, but you also can use any version of Windows, any Macintosh OS, or your Linux box to learn how to work with ASP and JavaScript.

To find a hosting service, use "Windows NT Hosting" as a keyword in any of the search engines, such as Excite, Yahoo! or AltaVista. Prices will vary widely, but because you do not need much server space for the short examples in this chapter, you can get the cheapest one available. To try it out, sign up for a service that has low (or no) setup fees as well as low monthly service fees. You don't need a domain name, just an IP address that the hosting service can provide. (An IP address to access an ASP file looks something like `http://323.64.12.543.23/hotJava/good.asp`*.) For example,* `www.hostek.com/plans. shtml` *has one plan with a monthly fee of $12.95 for 50MB of disk space with no setup fee. So, for about $13, you can spend a month learning ASP in a real-world environment to see if it's what you need. (The price quoted was at the time of this writing and might change.)*

Creating ASP Pages

An ASP script has a beginning tag, `<%`, and an ending tag, `%>`, that work like the containers in HTML. In some ASP pages, you will mix HTML and JavaScript or even some XML, but here the focus is on using VBScript as the main scripting language for the server-side script and using JavaScript for the client-side script. The basic ASP container using VBScript has the following format:

```
<%
```

```
VBScript
%>
```

Writing VBScripts

Use your favorite JavaScript text editor to create your ASP scripts. If you use Notepad, place quotation marks around the name of your file when you save it, and Notepad will add no unwanted .txt extension. The following script will get you started:

```
<%
Dim WhatzUP
WhatzUP="Where did I put my code?"
Response.write WhatzUP
%>
```

Save the file as whatzUP.asp in your server root directory or subdirectory. For example, I saved my ASP file in a folder (directory) named hotJava in my web server's root directory, active1.hartford.edu. To launch the program, I would type this:

```
http:// active1.hartford.edu /hotJava/whatzUP.asp
```

Use your domain and subdirectory names instead of `active1.hartford.edu` and `hotJava`. When you call up the script in your browser, you will see this in the browser window:

```
Where did I put my code?
```

As in JavaScript, if you get an error, check your code and check your directory.

Basic Screen Display Format

To display information to the screen, VBScript uses the statement `Response.write`, not unlike `document.write()` in JavaScript. The statement works like `document.write()` in that it accepts literals, variables, functions, or expressions. For example, this line:

```
Response.write "I like JavaScript."
```

works like this:

```
document.write("I like JavaScript.")'
```

Variables in VBScript

While declaring a variable in JavaScript is optional, it is not in VBScript. You must use a `Dim` (for dimension) statement to first declare the variable.

```
<%
Dim animal
animal = "Zebra"
%>
```

To declare multiple variables, you can use a single `Dim` statement:

```
Dim customers, products, prices
```

So, remember that while declaring a variable is optional in JavaScript, it is not in VBScript.

VBScript Data Types

As in JavaScript, data are not typed. That is, the variables are "smart" and will change type with the context of their use. Also as in JavaScript, you can use HTML tags as values in VBScript.

```
<%
Dim newpar, one, two, three
newpar = "<p>"
one = "The loneliest number."
two=222.222
three= 3
Response.write one
Response.write newpar
Response.write two
Response.write newpar
Response.write three
%>
```

The output of the previous example will be this:

```
The loneliest number
22.222
3
```

If you treat variables the same way in VBScript as you do in JavaScript, you should be fine. However, you must remember to use the `Dim` statement to declare all variables first.

VBScript Comments

Comments in VBScript work like those in JavaScript. They are ignored by the parser, but they inform the programmer. VBScript uses a single quotation mark (`'`) instead of double slashes (`//`); otherwise, they work the same. The following example shows a comment in VBScript:

```
<%
Dim customer
customer="Frank Talker"
'Each customer name is first placed into a variable.
```

```
Response.write customer
%>
```

Operators and Conditional Statements

You have to pay close attention to the operators in JavaScript and VBScript. Some are the same, and some are different. Critical differences can be found in the use of the assignment variable (=) in both assignment statements and conditional statements. Table 15.1 shows the operators used in VBScript.

Table 15.1. VBScript Operators

Operator	Use	Format Example
`=`	Assignment	`inven=832`
`+`	Addition	`total=item+tax+shipping`
`-`	Subtraction	`discount=regPrice-salePrice`
`*`	Multiplication	`itemTotal=item * units`
`/`	Division	`distrib=all/part`
`\`	Integer division	`hack=433.33\32`
`Mod`	Modulus	`leftOvr = 87 Mod 6`
`u`	Exponentiation	`cube = side ^ 3`
`&`	Concatenation	`FullName = "Java" & "Script"`
`=`	Equal to	`if alpha = beta Then ….`
`<>`	Not equal to	`if Jack <> Jill Then ….`
`>`	Greater than	`if elephant > mouse Then ….`
`<`	Less than	`if subtotal < 85 Then ….`
`>=`	Greater than or equal to	`if counter >= sumNow Then ….`
`<=`	Less than or equal to	`if 300 <= fullAmount Then ….`
`Not`	Negation	`if not (alpha > beta) Then….`
`And`	Logical AND	`if (alpha > beta) And (delta < gamma) Then ….`
`Or`	Logical OR	`if (alpha=beta) Or (delta=gamma) Then….`
`Xor`	Logical XOR	`if (sum >=1000) Xor (tax > .08) Then….`
`Eqv`	Logical equivalence	`if (alpha = beta) Eqv (delta=gamma) Then….`
`Imp`	Logical implication	`if alpha Imp beta Then….`

VBScript Conditionals

VBScript conditionals are similar to those found in the different versions of the Basic language, especially Microsoft's Visual Basic. If you are familiar with just about any kind of Basic programming, you will be on familiar ground. However, the logic of VBScript conditional statements is pretty much the same as that found in JavaScript, so you should not have too much difficulty.

The if/then Statement

VBScript employs this format:

```
if statement then another statement
```

which is equivalent to JavaScript's

```
if (statement) {
      another statement
}
```

The then keyword replaces the first curly brace, and there is no equivalent to the second curly brace. For example, the following script uses the if/then format in a single line:

```
<%
Dim alpha, beta
alpha="JavaScript"
beta="VBScript"
if alpha <> beta then Response.write "They are different."
%>
```

As in JavaScript, the else keyword can take the script to a different path when the condition is not met, as the following script shows:

```
<%
Dim designTime, payment
designTime=88
payment=5500
if payment > 5000 And designTime < 80 then Response.write "Take the
job." Else
Response.write "The schedule is too long."
%>
```

Using *ElseIf* and *Case*

The ElseIf keyword can be used when you have several different conditions. The last statement before End If is an Else statement. You can have as many ElseIf statements between the first If condition and the Else condition as you want.

```
<%
dim alpha, beta
alpha=10
beta=20
If alpha = beta then
      Response.write "They are equal"
ElseIf alpha > beta then
      Response.write "The first is bigger"
ElseIf alpha < beta then
      Response.write "The second is bigger!"
Else
      Response.write "I give up"
End If
%>
```

As in JavaScript, you might not prefer to use the `ElseIf` structure. VBScript has a similar statement to the `switch…case` statement in JavaScript. It has the following structure:

```
<%
Dim flowers
flowers = "rose"
Select Case flowers
Case "posey" Response.write "Two pence please."
Case "daisy" Response.write "Her last name was Miller."
Case "rose" Response.write "By any other name…."
Case Else Response.write "No flower of that nature is available."
End Select
%>
```

In the previous example, the screen would display this:

```
By any other name….
```

This is because the `Case` variable, `flowers`, is defined as `rose`, and the `Case` with `rose` prompts a `Response.write` of `By any other name….` Whichever structure you prefer, the `Elseif` or `Case`, is available for scripts with multiple conditions.

Determining Data Types in Conditional Statements

The `VarType()` function in VBScript is handy for determining what type of data is in a variable at any given time. This function returns the variable as 1 of 16 coded values. In the following example, a string variable and numeric variable are detected in a conditional statement:

```
<%
Dim stringVar, numVar, typeVar, newline
newline="<br>"
stringVar="I am a string."
numVar=1234
if VarType(stringVar) = 8 then Response.write stringVar & " This is a
string." &
newline
if VarType(numVar) = 2 then Response.write numVar & " This is an
integer."
%>
```

When the program executes, you will see the following on the screen:

```
I am a string. This is a string.
1234 This is an integer.
```

The conditional statement uses the codes in VBScript to sort the precise data type. Table 15.2 shows all of the codes.

Table 15.2. VarType Codes

Code	Data Type in Variable
0	Empty
1	Null
2	Integer
3	Long integer
4	Single
5	Double
6	Currency
7	Data
8	String
9	Object
10	Error
11	Boolean
12	Variant (arrays only)
13	Data access object
17	Byte
8192	Array

`TypeName()` is another useful function in VBScript for determining data type. However, instead of returning a number, it uses a string. The following script demonstrates some of the `TypeName()` syntax:

```
<%
dim alpha, beta, gammma, delta,newline
alpha=5
beta=5.7
gamma="Alfred"
delta=CDate(#06/06/2004#)
newline="<br>"
Response.write TypeName(alpha) & newline
Response.write TypeName(beta) & newline
Response.write TypeName(gamma) & newline
Response.write TypeName(delta) & newline
%>
```

When you launch the script, you will see the following data types listed:

```
Integer
Double
String
Date
```

Loop Structures

As in JavaScript, VBScript has more than one loop structure. The `for/next` loop is like the `for` loop in JavaScript, and the `do/while` loop is equivalent to the `do/while` loop in JavaScript. The `do/until` loop is like the `do` loop in JavaScript.

For/Next Structure

The `for/next` structure in VBScript expects a beginning and ending value for the loop. With each iteration of the loop, any statements between the `for` and `next` statements are executed. The following example shows a simple application using a loop to place formatted values on the screen. Note how the variable `spacer` is created using a loop as well.

```
<%
Dim counter, newline,spacer,x
newline="<br>"
for x=1 to 10
      spacer="-" & space
next
For counter = 1 to 25
      Response.write (counter & spacer & counter + 25& spacer &
counter + 50&
      spacer & counter + 75 & newline)
Next
%>
```

As with JavaScript, the `for/next` structure is best employed when the beginning and terminating values are known. With the two `do` loop structures, some kind of independent counter variable signals the end of the loop.

Do/While Loop

The `do/while` loop begins with the termination condition and keeps looping until that condition is met. The following example shows this loop at work:

```
<%
Dim counter, newline
newline="<br>"
counter=12
Do While counter >-1
      Response.write ( "Current dot com value is $" & counter & "
million.")
      Response.write newline
      counter = counter - 1
Loop
%>
```

You really need a script that tells you that your dotcom stock is going to drop *before* it goes through the floor!

Do/Until Loop

The `do/until` loop keeps repeating until the `stop` condition is met. It is the logical opposite of the `do/while` loop. (Using it, your dotcom stock will fare even worse!)

```
<%
Dim counter, newline
```

```
newline="<br>"
counter=12
Do Until counter >-1
      Response.write ( "Current dot com value is $" & counter & "
million.")
      Response.write newline
      counter = counter - 1
Loop
%>
```

The `Do Until` loop *gives one additional* iteration after it encounters the termination condition.

Arrays

Arrays in VBScript and JavaScript are the same in concept but are put together a little differently. The multidimensional arrays in VBScript are easier to work with than their counterpart in JavaScript. Like variables, arrays begin with a `Dim` statement, but an array size is provided as well. The following shows a simple example:

```
<%
Dim item(3)
item(0) = "Greater Swiss Mountain Dogs"
item(1) = "Bernese Mountain Dogs"
item(2) = "Pyrenean Mountain Dogs"
Response.write item(2)
%>
```

Note that, in VBScript, the array elements are in parentheses and not brackets, but, as in JavaScript, the first element is `0` and not `1`.

Multidimensional Arrays

If you are working with more than a single dimension and need to put it in an array in VBScript, you will find it easy to do, as is the case in most Basic programming languages. This statement has 51 elements in one dimension and 15 in another:

```
Dim SalesRep (51,15)
```

Using two dimensions, it might help to envision the array as follows:

```
Dim SalesRep (Row,Column)
```

A database with sales representatives in 50 states has up to 15 representatives in each state. (Fifty-one states are dimensioned because the `0` element will not be used in this particular example.) The states are identified by their admission to the union. Maryland has four representatives listed in the two-dimensional array in the following script:

```
<%
```

```
Dim SalesRep(51,15), newline, counter
newline="<br>"
salesRep(7,0)="Smith"
salesRep(7,1)="Jones"
salesRep(7,2)="Lee"
salesRep(7,3)="Hallohan"
salesRep(7,4)="Gonzalez"
for counter = 0 to 4
      Response.write salesRep(7,counter) & newline
next
%>
```

You should get the following output:

```
Smith
Jones
Lee
Hallohan
Gonzalez
```

Using a loop structure, as the previous example shows, finding what you want in the array is simple and keeps everything well organized.

Functions

VBScript has built-in functions and user functions just like JavaScript. (Microsoft keeps a list of the built-in functions at http://msdn.microsoft.com/scripting/default.htm?/scripting/vbscript.) You will find JavaScript and VBScript functions very similar. The user function has this format:

```
Function functionName(arguments)
      Statements
End function
```

The following shows a simple example and includes a built-in function, `Space()`, to add some space between the first and last names:

```
<%
Function fullName(lastName,firstName)
      Response.write firstName & Space(1) & lastName
End function
Response.write fullName("Sanders","Bill")
%>
```

Passing Data from JavaScript to ASP

In the role of a confirmation tool, JavaScript can check the content of forms before data is sent to an ASP page. Such a role is important for real-world forms because, if troublesome data are sent to an ASP page, the wrong data could end up in your database.

To see one role of JavaScript in gathering information, these next two scripts, one an HTML page and the other a VBScript script, show how data can originate in JavaScript and be passed to an ASP page.

Two key lines are used to pass data from HTML to ASP in the two scripts. On the HTML side, the `<form>` tag has two attributes, `action` and `method`. The method to pass data is either `post` or `get`. With ASP pages, use `post`. The action is to load the page specified in the URL assigned to the `action` attribute. Use the following general format:

```
<form method=post action="myServPag.asp">
```

On the server-side, use the VBScript function `Request.form("varName")`. The function expects an argument in quotation marks with the name of the variable from the HTML page. Using the `Request.form()` function, *you must remember to place quotes around the variable name.* Normally in a function in JavaScript, quotation marks are reserved for string literals, as is the case in VBScript. However, using `Request.form()`, you find an exception to this rule. The following general format is used:

```
<%
Dim varName
varName=Request.form("varName")
%>
```

The name of the variable in the ASP page can be any name that you want; however, I found that by using the same name in both the HTML and ASP pages, keeping track of everything is easier.

When you submit an HTML form, *all variables in the form* are passed to the ASP page. Therefore, you can have several variables sent at once, and each can be stored separately in a VBScript variable or array element.

The JavaScript function simply loads a hidden form with a value. Note that the JavaScript variable `greet` is placed into a hidden form named `alpha`. The variable that is passed to the ASP page script is `alpha` because ASP is getting its variable from a form element named `alpha`. Thus, JavaScript makes a “bank shot” to the form and then to ASP rather than directly to the VBScript script.

On the ASP side of the equation, you will see very little VBScript. However, you can use CSS to style your ASP page. In fact, you will see that most of the code is HTML or CSS, with just a few lines of VBScript.

getData.html

```
<html>
<head>
<title>Sending Data to an ASP Page </title>
<script language="JavaScript">
function loadIt() {
      var greet="Telegram from ASP!";
      document.storage.alpha.value=greet;
      }
</script>
```

```
</head>
<body onload="loadIt()";>
<form name="storage" method=post action="greetings.asp">
      <input type=hidden name="alpha" >
      <input type=submit value="Submit form and get this page's
message back from ASP">
</form>
</body>
</html>
```

Save this script as getData.html. It uses the `post` method to open an ASP page named greetings.asp. The ASP page immediately echoes whatever is in a variable named `alpha`. As you will see, because `alpha` is not declared or defined on the ASP page, its content could be only what has been passed to it from the HTML page. Save the following page as greetings.asp, and put it in the root directory or subdirectory in your root directory along with the getData.html page.

greetings.asp

```
<html>
<head>
<style type="text/css">
body {
      font-family:verdana;
      font-size:24pt;
      color:#ffff33;
      background-color:#f3cc00;
      font-weight:bold;
      }
#blackground {
     background-color:black;
     }
</style>
</head>
<body>
<div ID=blackground>
      <strong>
            <center>
                   <%
                   Dim alpha
                   alpha=Request.form("alpha")
                   Response.write alpha
                   %>
            </center>
      </strong>
</div>
</body>
</html>
```

When you run the getData.html script, press the Submit button to launch the ASP script. Your screen shows the message "Telegram from ASP!"

Controlling Multiple ASP Pages with JavaScript

One use of JavaScript is to serve as a controller for ASP pages appearing in an HTML environment. Because ASP replaces a page that it is called from, one plan of action is to use JavaScript to create a number of buttons used to launch different ASP pages within a frameset. JavaScript can reside in the menu page, and ASP is called in the body page of a two-column frameset. At the same time, both the JavaScript page and the ASP page share a common external style sheet. Seven scripts are required for this project: an external CSS file, a frameset page, the menu page using JavaScript, a placeholder page, and three ASP pages demonstrating different features of ASP.

Cascading Style Sheet

Using a color scheme from *The Designer's Guide to Color Combinations by Leslie Cabarga* (North Light Books, 1999), the first step is to create a palette using the following colors (all values are in hexadecimal):

Color A	FFE699	Light tan
Color B	CC994C	Tan
Color C	CC0019	Deep red
Color D	00804C	Dark green
Color E	808080	Gray
Color F	000000	Black

This particular color scheme was selected because it contains black and gray, the colors of the button in HTML forms. The following style sheet incorporates the colors used in both the ASP and HTML pages.

controller.css

```
.bigRed {
       color: #cc0019;
       background-color: #ffe699;
       font-family: verdana;
       font-weight: bold;
       font-size: 18pt;
       }
.midRed {
         color: #ffe699;
         background-color: #cc0019;
         font-family: verdana;
         font-weight:bold;
         font-size: 14pt;
         }
```

Save the CSS style sheet in the same directory as the files for the HTML and ASP pages. All of the pages will employ the same style sheet.

This next HTML page establishes the frameset where the initial HTML pages—and, eventually, the ASP pages—will go.

controlSet.html

```
<html>
<frameset cols="25%,*" border=0>
      <frame name="menu" src="jMenu.html" frameborder=0 scrolling=0>
      <frame name="display" src="jsD.html" frameborder=0 scrolling=0>
</frameset>
</html>
```

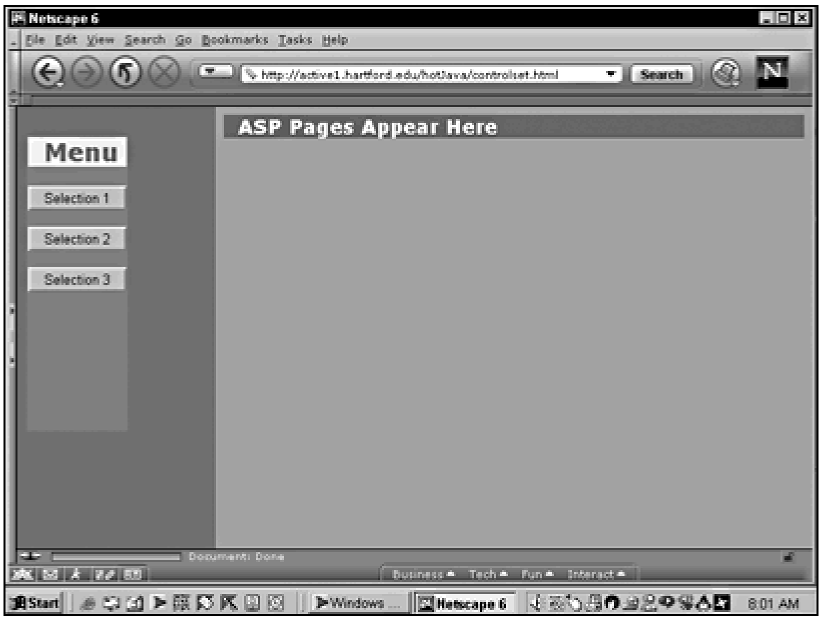
The left side of the frameset opens with a menu page and a blank HTML page in the right frame. All of the JavaScript controllers are in the menu. A simple function in JavaScript is used to launch any of the ASP pages in the right frame.

jMenu.html

```
<html>
<head>
<link rel="stylesheet" href="controller.css" type="text/css">
<script language="JavaScript">
      function loadASP(url) {
      parent.display.location.href=url;
      }
</script>
</head>
<body bgcolor=#00804c>
&nbsp;
<table BORDER=0 height=70%>
      <tr align=left valign=top>
            <td bgcolor="#808080"><center>
                  <div class="bigRed">&nbsp; Menu&nbsp;</div>
                  </center>
                  <br><form>
                        <input type=button
onClick="loadASP('alpha.asp')"
value="Selection1">
                        <p><input type=button
onClick="loadASP('beta.asp')"
value="Selection2">
                          <p><input type=button
onClick="loadASP('gamma.asp')"value="Selection 3">
                     </td>
               </tr>
</table>
</body>
</html>
```

The initial page in the right frame is a "dummy" page used to occupy space until one of the ASP pages is launched. Figure 15.1 shows how the initial page appears when the frameset is loaded.

Figure 15.1. The area on the right is reserved for ASP by an initial placeholder page.


Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://active1.hartford.edu/hotJava/controlset.html
Search
Menu
Selection 1
Selection 2
Selection 3
Document: Done
Business Tech Fun Interact
Start
Windows
Netscape 6
8:01 AM

ASP Pages Appear Here

jsD.html

```
<html>
<head> <title>Display Page</title>
<link rel="stylesheet" href="controller.css" type="text/css">
</head>
<body bgcolor=#cc994c>
      <div class=midRed> &nbsp ASP Pages Appear Here &nbsp </div>
</body>
</html>
```

The first ASP page simply displays a message in plain text. However, it does load the same style sheet as the two previous HTML pages.

alpha.ASP

```
<html>
<head> <title>Control Menu</title>
<link rel="stylesheet" href="controller.css" type="text/css">
</head>
<body bgcolor=#808080>
<center>
<%
Response.write "This is plain text."
%>
</center>
</body>
</html>
```

The second ASP page responds with a text message using one of the style sheet classes.

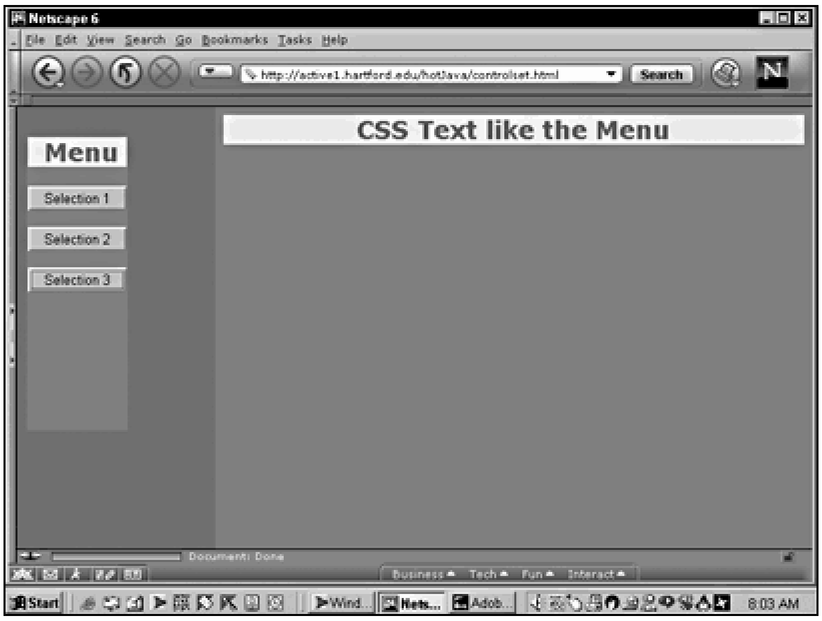
beta.ASP

```
<html>
<head> <title>Control Menu</title>
<link rel="stylesheet" href="controller.css" type="text/css">
</head>
<body bgcolor=#808080>
<center>
```

```
<%
Response.write "<div class=midRed>&nbsp Here's the other CSS class
&nbsp </div>"
%>
</center>
</body>
</html>
```

The third ASP page, like the second, responds with a message that indicates that yet another of the CSS classes has been employed (see Figure 15.2).

Figure 15.2. An ASP page using the same CSS style sheet as the HTML page appears in the right column.



gamma.ASP

```
<html>
<head> <title>Control Menu</title>
<link rel="stylesheet" href="controller.css" type="text/css">
</head>
<body bgcolor=#808080>
<center>
<%
Response.write "<div class=bigRed>&nbsp CSS Text like the Menu
&nbsp</div>"
%>
</center>
</body>
</html>
```

Microsoft Access, ASP, and JavaScript

One of the most popular Windows OS database programs in use is Microsoft's Access. Using *Access 2000* , this section shows how to use HTML and JavaScript as a front end to display, search for, and add records to an *Access 2000* file on a server. Most of the work is done using VBScript and Structured Query Language (SQL) commands to query an *Access 2000* file on a server using variables in a `script` for showing what is in the database and as an input source for adding new records. The database is made up of the following nine fields:

- Identity number
- Last name
- First name
- Address
- City
- State
- ZIP code
- Phone
- Email

The purpose is to create a table to show the information of a list of people stored in the database. Each field must include a name, a data type, and the length of the field for fixed-length types.

Three different files are involved in sending data between an HTML page and a database.

Setting Up the Access 2000 File

You will need *Microsoft Access 2000* for the rest of this chapter. All of the example files were created using the Windows version of Access; at this time, Microsoft is not selling a Macintosh version of Access. However, the Access files on the book's web site can be used on an NT Server, and all of the ASP pages created with the Macintosh can read and write the files and return data from the files.

For Windows users who have Access, the following set of steps shows how to set up the database. (The steps were written using *Access 2000* , but older versions can be used as well, with some slight differences. The point is to create a .mdb file with the set of fields shown.)

1. Launch *Access 2000* from the Start menu. Select Blank Access database in the dialog box, and click OK.
2. In the File New Database window, first select the desktop as the folder, and then use the filename javaS.mdb in the File Name text window at the bottom of the window. Click Create.
3. Double-click the option Create Table in Design View in the Database window. In the Design View window, enter the following in the Field Name and Data Type columns. Use the pop-up menu in the Data Type column to select the data type. (All of the nine fields are text except for the first, which is AutoNumber.) Unless otherwise noted, all of the field sizes are the default size.

- **ID** AutoNumber. (Click the Key icon on the toolbar to make this field the primary key.)
- **Lname** Text (default).
- **Fname** Text (default).
- **Address** Text (default).
- **City** Text (default).
- **State** Text (default); field size 2.
- **Zip** Text (default); field size 5.
- **Phone** Text (default); field size 10.

- **Email** Text (default).

4. After you define the nine fields, click the Disk icon on the toolbar and save the table using the name JavaStable. Next, select File, Close from the menu bar.
5. When you close the Design View window, you will see a window with your new table in the javaS: Database window. Double-click the JavaStable icon to open it. Enter three names and the rest of the fields in the table. The ID column automatically provides a unique ID number as you enter data into the other two fields.
6. After entering the data, click the Save icon (disk in the toolbar) and then select File, Exit. You're finished with entering data directly into the *Access 2000* database table and file.

Placing the Access 2000 File on the Server and Preparing the DSN

The next step is to open the root folder where you have been placing your ASP pages and place your database file. If you are using a remote server, just use FTP to move the *Access 2000* file to the root folder or subfolder on the NT Server.

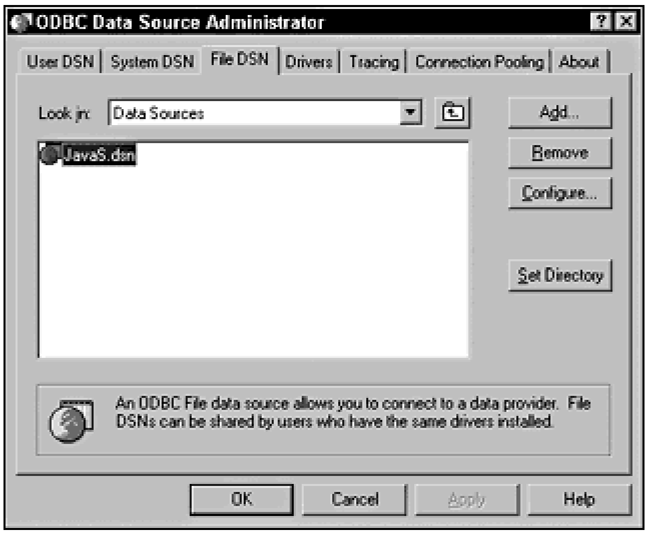
Next, you need to create a data source name (DSN) connection to the database on the server. This connection allows your ASP application to interact with the database. For *Windows 2000* and *Windows NT* , use the following steps.

(*Note*: These steps are on the server. If you are using your computer as both client and server, then you can follow these steps on your computer. If you are administering another NT server, these steps should be on the remote server.)

1. Select Start, Settings, Control Panel. In the Control Panel, double-click on the ODBC Data Sources icon to open the ODBC Data Source Administrator.
2. Click on the File DSN tab and click Add. Then select Microsoft Access Driver (*mdb) from the menu and click Next.
3. Type in a descriptive name. We used JavaS for the file DSN name. Click Next.
4. Click Finish, and then click Select. Navigate to the directory of your database file. It will appear in the left window under Database Name. Click on your database (for example, javaS.mdb) to select it, and then click OK.
5. Click OK *twice.* In the ODBC Data Source Administrator window in the File DSN tab, you will see your new DSN name with the extension .dsn added.

Figure 15.3 shows what you should see in the File DSN folder in the ODBC Data Source Administrator when you have correctly configured your data file.

Figure 15.3. Establishing your database file with an appropriate data source is essential when you use your own server.



An ODBC File data source allows you to connect to a data provider. File DSNs can be shared by users who have the same drivers installed.

When you enter the database filename in an ASP script, the DSN name is used by the server software to enable a connection.

Making the Connection Between Your ASP Page and Database File

To make a connection between an ASP page and a database, you need to know the path to the database. The process itself is simple enough *if you have the right path.* However, when you get the formula, you can reuse it on any database within the same server. The following path was used:

```
d:\Inetpub\Wwwroot\hotJava\javaS.mdb;
```

In the example, drive D is used, but the correct drive could have been drives C or E, or any other drive. The important fact to note in the path is that it is an *internal path* to the root directories on the server. So, while a different address had to be used to access the ASP pages (such as `http://myserver.com`) that contained the code, the path in the ASP page references a local, internal path for the server.

The path is just part of a longer driver definition required to connect the ASP script to the database and the data within it. Generally, the path is defined in a variable, and then the VBScript uses that variable in other commands to make the connection. First, the provider needs to be defined. The following *single line* defines the driver in the variable named `hookUp`:

```
hookUp= "Driver={Microsoft Access Driver (*.mdb)};
DBQ=d:\Inetpub\Wwwroot\hotJava\javaS.mdb;"
```

Because the provider information has been placed into a variable, when the provider has to be used, you need to use only the variable name instead of the long line of code.

The rudiments of setting up an ASP page to make a connection to a database and pull out data can best be seen in a complete page and example. In that way, you will be better able to understand the context of the required code. The following

ASP page pulls together the different parts needed to see the contents of a database on a server.

Reading an Access 2000 Database with ASP

After the *Access 2000* database has been set up and a DSN has been established for it, you are ready to create an ASP page to pull data from the Access file and send it to JavaScript. To begin, enter the following ASP script and save it in the root directory or a subdirectory within the root directory of your server. Save the file using the name readDB1.asp.

readDB1.asp

```
<%
Dim hookUp, Conn, ViewRecord, Lname, output, display, SQL
hookUp= "Driver={Microsoft Access Driver (*.mdb)};
DBQ=d:\Inetpub\Wwwroot\hotJava\javaS.mdb;" Set Conn =
Server.CreateObject("ADODB.Connection")
Set ViewRecord = Server.CreateObject("ADODB.Recordset")

Conn.Open hookUp
SQL="SELECT * FROM JavaStable"
ViewRecord.Open SQL, Conn

Do While Not ViewRecord.EOF
      Lname=ViewRecord("Lname") & "<br>"
      display =display & Lname
      ViewRecord.MoveNext
Loop
Response.Write display
ViewRecord.Close
Conn.Close
Set ViewRecord = Nothing
Set Conn = Nothing
%>
```

The first part of the script is for dimensioning the variables, defining the connections, and then setting the connections. The several variables (such as `hookUp` and `Conn`) are declared using the `Dim` statement. Next, the driver and provider are placed into a variable named `hookUp`. The `ADODB` connection object is placed into the variable `Conn`, and the `ADODB recordset` object is placed into a variable named `ViewRecord`.

The next part opens the connection and defines an SQL command. The `action` query selects all the records (* = wildcard) from the table named JavaStable in the *Access 2000* file named javaS.mdb. The next line opens the recordset in the database defined in the connection and issues the SQL command.

One of the nicer features of VBScript is that it can loop through a recordset until it encounters the `EOF` (end of file) of the recordset. Note also the use of the `<br>` tag in this block. Used in HTML as a line break, you will find that, in a text field, the break works as well. The `display` variable concatenates the data with the line break tag through the entire loop. To keep this first example simple, only one of the nine fields, `Lname`, is passed to a variable, `display`, that collects all of the data for the first field.

Next, the `display` variable is used with `Response.Write` to send the data to the screen. Finally, the open connections are closed and the variables are reset to `Nothing`.

Reading and Displaying Multiple Fields

The first script showed only one field, to simplify the process and focus on what needed to be done for connection. The following script looks at all nine fields and all the records in the fields. Save the script in your root directory or subdirectory in the root directory.

ReadDB2.asp

```
<%
Dim hookUp, Conn, ViewRecord, ID, Lname, Fname, Address, City, State,
Zip, Phone, Email,
display, SQL
hookUp= "Driver={Microsoft Access Driver (*.mdb)};
DBQ=d:\Inetpub\Wwwroot\hotJava\javaS.mdb;"
Set Conn = Server.CreateObject("ADODB.Connection")
Set ViewRecord = Server.CreateObject("ADODB.Recordset")

Conn.Open hookUp
SQL="SELECT * FROM JavaStable"
ViewRecord.Open SQL, Conn
Do While Not ViewRecord.EOF
      ID=ViewRecord("ID")
      Lname=ViewRecord("Lname")
      Fname=ViewRecord("Fname")
      Address=ViewRecord("Address")
      City=ViewRecord("City")
      State=ViewRecord("State")
      Zip=ViewRecord("Zip")
      Phone=ViewRecord("Phone")
      Email=ViewRecord("Email")
      display =display & ID & "<br>" & Fname & Space(1) & Lname &
"<br>" & Address &
"<br>"
      & City & "," & Space(1) & State & Space(1) & Zip &  "<br>" &
Phone & "<br>" & Email
      & "<p>"
      ViewRecord.MoveNext
Loop
 Response.Write display

ViewRecord.Close
Conn.Close
Set ViewRecord = Nothing
Set Conn = Nothing
%>
```

All of the variables use the same name as the fields in the database. In that way, you are less likely to confuse what is what.

Inserting Records into Access from HTML

The final step is to create an HTML page through which you can add data to your database. JavaScript plays the role of checking your data before submitting it.

First, you need to create your web page for data entry, and then you need to create your ASP page to send the data to your Access file on the server. (After you add your data, you can use the script in the previous section to read it and make sure that it's all there.)

addRecords.html

```
<html>
<head>
<style type="text/css">
body {
      font-family:verdana;
      font-size:11pt;
      font-weight:bold;
      background-color:ffabab;
      }
#myText {color:ff2626; background-color:black}
</style>
<script language="JavaScript">
function verify() {
      var flag=0;
      dv=document.reporter;
      for(var counter=0;counter < dv.length-2;counter++) {
            if(dv.elements[counter].value=="") {
            flag=1;
      }
      }
      if(flag==1) {
            alert("Please fill in all parts of the form:");
     } else {
            alert("Form is ready to submit");
      }
}
</script>
</head>
<body onLoad="document.reporter.reset()">
<h3>Fill in all windows in the form </h3>
<div ID="myText">
<form name="reporter" method=POST
action="http://active1.hartford.edu/hotJava/addRecords.asp">
      <input type=text name="ID">ID <br>
      <input type=text name="Lname">Last Name <br>
      <input type=text name="Fname">First Name <br>
      <input type=text name="Address"> Address <br>
      <input type=text name="City"> City <br>
      <input type=text name="State" size=2>State <br>
      <input type=text name="Zip" size=5>Zip <br>
      <input type=text name="Phone" size=10> Phone <br>
      <input type=text name="Email"> Email <p>
      <input type=button value="Verify:" onClick="verify()"> &nbsp;
      <input type=submit value="Send information to database:">
</form>
</div>
</body>
</html>
```

The script for inserting data into the database must access a file on the NT server. An added HTML tag is used to access an ADO (Active Data Objects) file required when you add data. Using a set of double and single quotations, the SQL variable (and, thereby, command sequence) is set up; when that's done, the script follows a familiar path.

addRecords.asp

```
<!-- METADATA TYPE="typelib"
              FILE="C:\Program Files\Common Files\System\ado\
msado15.dll" --> <HTML>
<%
Dim Conn,Cmd,
intNoOfRecords,jsID,jsLname,jsFname,jsAddress,jsCity,jsState,jsZip,js
Phone,jsEmail,SQL
jsID=Request.Form("ID")
jsLname=Request.Form("Lname")
jsFname=Request.Form("Fname")
jsAddress=Request.Form("Address")
jsCity=Request.Form("City")
jsState=Request.Form("State")
jsZip=Request.Form("Zip")
jsPhone=Request.Form("Phone")
jsEmail=Request.Form("Email")
Conn = "Provider=Microsoft.Jet.OLEDB.4.0;" & "Data
ÂSource=d:\Inetpub\Wwwroot\hotJava\javaS.mdb;"
Set Cmd = Server.CreateObject("ADODB.Command")
Cmd.ActiveConnection = Conn
SQL= "INSERT INTO javaStable(ID,
Lname,Fname,Address,City,State,Zip,Phone,Email) VALUES
(" & jsID & ",'" & jsLname & "','"& jsFname & "','" & jsAddress &
"','" & jsCity & "','"
& jsState & "','" &
jsZip & "','" & jsPhone & "','" & jsEmail & "')"
Cmd.CommandText =SQL
Cmd.CommandType = adCmdText
Cmd.Execute intNoOfRecords
Set Cmd = Nothing
Response.Write "Records entered:"
%>
</HTML>
```

When they're in the VBScript, the variables are dimensioned and then data from the HTML page is pulled into the page using `Request.Form()`. The nine variables matching the HTML names are inserted as ASP variables using a lowercase `js` before each of the HTML variable names. Next the connection protocols establish a link with the database. (If you are using *Access 97* , substitute 4.0 with 3.1 right after OLEDB.)

Using the SQL command `INSERT`, you list the names of each field and then insert data into them. If you used literals, your `VALUES` would look like this:

```
"…VALUES (21, 'Adams','John','32 Bath', 'Boston', 'MA', '05432','322-
123-4567'
'jadams@tparty.gov')"
```

Because you need to use variables instead of numeric and text literals, the variables must be concatenated into the format recognized by VBScript. Everything but the variables themselves needs to be in quotation marks, and all text must have single quotation marks around it. When copying the SQL contents, be very careful.

Summary

As with PHP, JavaScript can play a verification role with ASP pages. A good deal of the scripting work is handled by VBScript, and, on the server side of the equation, that is to be expected. However, as you delve deeper into back-end scripts, you will find that more use can be found in a good JavaScript verification script on the client side.

Much more can be done with both JavaScript and ASP pages than could be included in this chapter. However, the point is that you can effectively use the two together to gather data, send the data to an Access database, and then view the data stored. As a starting point, this chapter should have given you a clear idea of how ASP pages can be effectively used with HTML, JavaScript, and CSS.

Chapter 16. CGI and Perl

CONTENTS>>

Like other server-side languages, the Practical Extraction and Report Language (Perl) is associated with a protocol, the Common Gateway Interface (CGI). Other languages are used with CGI as well but, just as VBScript is the main language associated with Active Server Pages (discussed in the previous chapter), Perl is the main language associated with CGI.

While direct scripting can occur between a UNIX (or Linux) server and your computer, the material presented in this chapter assumes that most readers want to know how to create scripts with a Windows or Macintosh OS, and not Telnet or work from a UNIX shell. So, the focus will be on writing Perl scripts and examining how they interface with JavaScript and HTML pages. The major scripting tool (if you can call it that) will be the humble text editor.

Scripting with Perl

Like all scripting languages, Perl has a set of statements, variables, functions, operators, and other typical language features. However, because Perl was designed to work with data to be passed between clients and servers, it has special commands for handling data to be stored and retrieved from the server. Variables in HTML forms can be passed to a Perl script, and JavaScript's main role lies in preprocessing the variables in the forms.

Getting Started

You will need a hosting service that provides you with access to CGI and a Perl interpreter. The examples used for this book were done using JTLNet as a hosting service (`www.jtlnet.com`), and you will find them to have everything you need for simple access for CGI. If you already have a hosting service, it should be capable of showing you how to set up and use CGI. If your service does not allow CGI, get another hosting service that does. You can find plenty of high-quality services for less than $8 per month.

When you get squared away with a hosting service, you will need to know where to store your CGI scripts written in Perl. Usually, the name of the folder (directory) where you put your scripts is cgi-bin. Use your favorite FTP program to place your Perl scripts in the cgi-bin directory using ASCII (*not* binary) transfer. You will find it in the root HTML directory, which is usually named something like public_html. The path to your Perl scripts, which end with the extension .pl, would be as follows:

```
http://www.domain.com/cgi-bin/scriptName.pl
```

The directory public_html (or whatever the root HTML directory is called) is not named in the path because it is the *root* for your domain for HTML.

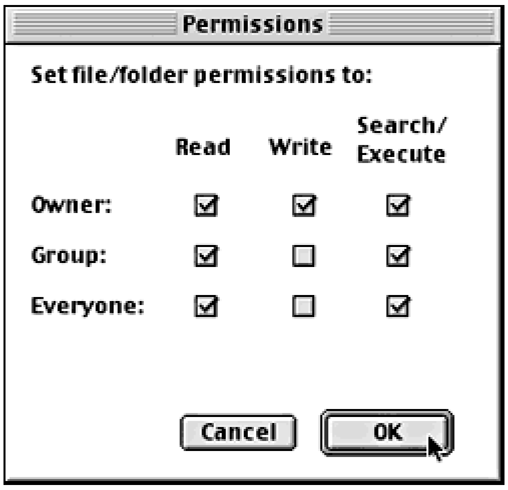
chmod

Unlike PHP or ASP pages, CGI scripts need to be configured when they are on the server in the cgi-bin folder. The UNIX command `chmod` changes the permission *for each Perl file* that you want to execute. (In other words, all your Perl files need to be configured.) Within a UNIX shell, the command to configure your script is this:

```
chmod 755 scriptName.pl
```

If you're using a typical Windows or Macintosh OS computer, you will need another way to set the permissions. You need an FTP application that can issue a `chmod` command or set the permission parameters in some other way. Both PCs and Macs have plenty of FTP programs that will do that. A popular FTP program for Windows is called CuteFTP (www.cuteftp.com), and it has its own `chmod` command built into a drop-down menu. You simply select the file that is on the server that you want to configure and select Commands, File Actions, CHMOD; then enter the 755 code in the Manual window. On the Mac side, the popular Fetch FTP program has a permissions option that you can set without having to write the `chmod` command. Selecting Remote, Set Permissions, you select all of the permission options *except* Group and Everyone Write. Figure 16.1 shows how it looks when completed for the 755 `chmod` setting.

Figure 16.1. You can click the permission options to the equivalent of a `chmod` command with some FTP applications.


Permissions

Set file/folder permissions to:

	Read	Write	Search/ Execute
Owner:	☑	☑	☑
Group:	☑	☐	☑
Everyone:	☑	☐	☑

A Quick Script

To get a quick overview of a running CGI script written in Perl, the following set of steps takes you through the sequence that you will need to get a working script:

1. Open up your favorite text editor. For the purposes of this book and most Perl applications with CGI pages, you begin all Perl scripts with this:
2.
```
#!/usr/bin/perl
```
3. Because you will be writing Perl scripts in pages that will generally go to the web, you will need another generic line of code:
4.
```
print "Content-type:text/html\n\n";
```
   The line does not print to the screen, but it will make the rest of the code do so.
5. Next, for output, you use a `print` statement to send a message to the screen. Just as in JavaScript, lines terminate with a semicolon, except that in Perl ending semicolons are not optional. You have to include them. (The `\n` simply puts a new line in the source of the HTML file.)
6.
```
print "I like JavaScript and Perl.\n";
```
7. Tell the parser that you've reached the end using the `exit` statement.
8.
```
exit;
```
9. At this point, the script is done and should look like the following:
10.
```
11. #!/usr/bin/perl
12. print "Content-type:text/html\n\n";
13. print "I like JavaScript and Perl.\n";
    exit;
```
   Save it as perlOne.pl.

14. Using an FTP program, place perlOne.pl in your cgi-bin directory (or your hosting service's equivalent).
15. When it is in the `cgi-bin` folder, select the file and, using the `chmod` command or permission matrix, configure the file to `chmod 755`.
16. Open your browser and, in the URL window, type in the following, substituting your own domain name:
17.

    ```
    http://www.domain.com/cgi-bin/perlOne.pl
    ```

    On the screen, you should see

    ```
    I like JavaScript and Perl
    ```

Other than having to initialize the script using `chmod` and adding the line for web pages, the process is not unlike loading a JavaScript file onto a server.

A Brief Perl Tutorial

JavaScript and Perl have some common features in terms of general structure, but Perl is unique and very powerful. One of the more powerful structures within Perl is a pattern-matching feature called regular expressions. JavaScript, too, has regular expressions; by comparing the two, you can learn about both. However, before getting to regular expressions, a quick overview of Perl is in order.

Unique Variable Characters of Perl

Perl distinguishes between different types of variables using a set of symbols with their own names. They include the following:

- **$** Scalar: a single number or string
- **@** Array: a multiple-item variable
- **%** Hash: a paired array

A scalar is similar to a plain-vanilla variable in JavaScript.

Scalar Variables

The scalar variables in Perl look a lot like variables in PHP because they begin with a dollar sign. In Perl, they are simply variables with a single value. (In some versions of Basic, all string variables begin with dollar signs, but both strings and nonstrings can be Perl scalar variables.) The following are examples of scalar variables—note that they include strings and numbers:

```
$subtotal = 22.33;
$membership=453;
$customer="Linda Sheppard";
```

One of the advantages of having identifying characters at the beginning of variables in Perl is that they can be put inside a literal without any special set of quotes or concatenation.

```
$meetingTotal=64;
$report="The meeting was attended by $meetingTotal last week.";
```

The advantage of not having to fiddle around with concatenation turns into a disadvantage when you are faced with having to use one of the variable characters such as the dollar sign ($). As in JavaScript, the dollar sign can be "escaped" by using a *backslash.* For example, the following variable assignment uses an escaped dollar sign:

```
$total="\$23.33";
print $total;
```

The output would be `$23.33`.

Arrays and Hashes

An array in Perl can be remembered by the `@` symbol ("at" sign for "array"—get it?). An array is declared by typing in the array name followed by the list of array elements in parentheses. For example, the following array contains several types of dogs:

```
@dogs = ("lab","sheltie","Swissy","mutt");
```

If you wanted to print out `Swissy`, you would enter this:

```
print $dogs[2];
```

At first glance, the `print` command looks like a typo, but it's not. In Perl, you use a scalar variable to access the elements of an array. All of the elements must be stated as follows:

```
$arrayName[n];
```

Because Perl variables are conceived of as singularities or pluralities, it makes sense to reference single elements of an array using the array name but a scalar symbol.

A unique variable format in Perl is the *hash.* Hashes are paired elements in an array or associative arrays. Hashes have two formats. The first format looks like an array, but the second is clearly descriptive:

```
%dogs = ("lab", "Labrador Retriever", "Berner", "Bernese Mountain
Dog", "Swissy",
"Greater Swiss Mountain Dog");
```

The first element is associated with the second, the third with the fourth, and so on. A clearer way of presenting the same hash follows:

```
%dogs= (
"lab" => "Labrador Retriever",
"Berner"=> "Bernese Mountain Dog",
"Swissy"=> "Greater Swiss Mountain Dog",
);
```

When you have your data in a hash, how do you get it out? By referencing the first of an associative pair, you get the second of the two. The format is

```
$hashName{"FirstElementOfPair"};
```

Your return is the second element in a hash pair. For example, using the dog example hash, if you typed this:

```
print $dogs{"Berner"};
```

your screen would show

```
Bernese Mountain Dog
```

Using the hash structure is handy when you have associative pairs and can reference one for the other.

Perl Comments

Comments in Perl are prefaced by pound signs (#). As in JavaScript, comments can be put on separate lines or in lines with other code. The following examples show how they can be used:

```
$alpha="News"; #This is a scalar variable.
@happy=("glad", "exuberant", "cheerful", "blithe"); #Synonyms for
happy;
#All comments are ignored by parser.
```

Perl Operators

Perl operators, for the most part, are identical to those in JavaScript, with some notable exceptions. First, numeric and string comparison operators are different. This difference can be very frustrating after using JavaScript because it’s easy to forget and use the wrong one. For example, the following script shows the correct way to use numeric and string operators:

```
$alpha="Apples";
$omega="Oranges";
if($alpha ne $omega) { #Uses a string comparison
      print "They are different \n\n";
      }
```

The common error in Perl occurs when the programmer uses a line like this when both scalar variables are strings:

```
if($alpha != $omega) { #Wrong!
```

So, when you write conditional statements in Perl, which are the same as in JavaScript, watch out for mixing string and numeric comparison operators. Table 16.1 shows the main operators in Perl. (File test operators are not included.)

Table 16.1. String and Numeric Operators in Perl

Operator	Use	Format Example
=	Assignment	$inven =832;
+	Add	$total = $item + $tax + $shipping;
-	Subtract	$discount = $regPrice - $salePrice
*	Multiply	$itemTotal = $item * $units;
/	Divide	$distrib = $all / $part;
==	Compare evaluation	if ($quarter1 == $quarter2)
!=	Not equal to	if (999 != $little) {
>	Greater than	if ($elephant > $mouse) {
<	Less than	if ($subTotal < 85) {
>=	Greater than or equal to	if ($counter >= 200) {
<=	Less than or equal to	if (300 <= $fullAmount) {
<=>	Comparison	$alpha=($beta <=> $delta);
eq	String compare evaluation	if ($Bob eq $Ben) {
ne	String not equal to	if ($big ne $little) {
lt	String less than	if ($art lt $science) {
gt	String greater than	if ("Big" gt "Tiny") {
le	String less than or equal to	if ("Apes" lt "Zebras") {
ge	String greater than or equal to	If ($first ge "Last") {
cmp	String comparison	$beta="dogs" cmp "cats";
+=	Compound assign add	$total += 21;
-=	Compound assign subtract	$discount -= (.20 * $item);
.	Concatenation	$wholeName = $firstName.$lastName;
.=	Compound concatenation	$areaCode .= $phone
&&	Logical AND	if ($first == 97 && $second <= $third) {
||	Logical OR	if ($high === 22 || $low == 12){
++	Increment (pre or post)	for ($la=6; $la <=78; ++$la)
--	Decrement (pre or post)	for ($ls=50; $ls >=12; $ls--)

Perl Statements

This brief section covers the main Perl statements and structures that you will use in a typical CGI page. Most of the statements work the same as they do in JavaScript, but the variables must be prefaced with the correct symbols, and some of the formats might be slightly different.

Output

As you have seen in examples in this chapter, the `print` statement seems to be the most common output statement. It is. Three key types of `print` statements will help you get started in Perl with CGI:

- Print literal: **`print "Hamster \n\n";`**
- Print variable: **`print $bigshot;`**
- Print HTML: **`print "<strong> Tough </strong>";`**

A print statement that outputs HTML tags such as `<b>` is really a formatting technique in CGI, just as using the backslash with `n` (`\n`) is formatting code.

When printing with variables and HTML tags, you can place the variable right in the middle of a `print` statement in quotation marks, and the variable is still recognized as such. For example, this next script segment would print `She is bold and beautiful` in boldface type:

```
$ugly = "beautiful";
print "<b>She is bold and $ugly .</b>";
```

Conditional Statements

Perl's conditional structures are very similar to those in JavaScript. The `if`, `if` / `else`, and `if` / `elsif` / `else` statements are almost identical, except that the key word `elsif` is spelled without an e. The following shows some conditional statements in a Perl script:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
$alpha="Angels";
$beta="clouds";
$gamma="sky";
if (($alpha ne $beta) && ($gamma gt $beta)) {
      print "$alpha live in $beta up in the $gamma .";
      } else {
      print "Where did all the angels go?";
      }
exit;
```

A *big difference* between Perl and JavaScript in conditional statements is that you *must* remember to use the numeric and string comparison operators correctly. In the previous example, note that the operators `ne` and `gt` were used instead of `!=` and `>`, as would be done in JavaScript.

A different conditional structure that you will find in Perl but not JavaScript is the `unless` statement. It is the *opposite* of the `if` statement. When the condition is *not true,* the statement in the `unless` conditional executes, as the following shows:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
```

```
$myDog="WillDe";
$goodDog="WillDe";
unless( $myDog ne $goodDog) {
      print "Sit. Stay. Play the piano."; #This will appear on the
screen.
      }
exit;
```

When going through a list of choices, each with a different outcome, use `elsif`. With CGI, this could be a common occurrence because several different values could be pulled out of a data file. The following example shows how `elsif` could be employed:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
$nmLast="Smith";
$nmFirst="Karen";
if($nmLast eq "Smith" && $nmFirst eq "Ed") {
      print "Hi Ed! How\'s the Smith family?";
      }
elsif ($nmLast eq "Smith" && $nmFirst eq "Jacqueline") {
      print "Hi Jackie! How\'s the Smith Flower Shop doing?";
      }
elsif ($nmLast eq "Smith" && $nmFirst eq "Ralph") {
     print "Hi Ralph! Congratulations on early parole!";
} else {
       print "Gee, I don\'t think we\'ve met. Would you like to
register with our
       site?";
}
exit;
```

In the example, the values for the two variables were assigned. However, the same script could be used if pulling data out of a text-data file.

Loops

Loops are important in Perl for searching through data files, as you will see in the next section. Loops in Perl and JavaScript share a good deal in common but, as with conditionals, you will see differences as well. The `for` loop has the same structure as the `for` loop in JavaScript:

```
for($counter;$counter < $min;$counter++) {
      statements…
      }
```

Instead of using the `for` keyword, you can substitute `foreach`. They work identically, but one of the two might be clearer. For example, the following goes through a list and prints it to the screen:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
foreach $customer("Tom","Dick","Harry") {
      print "$customer<br>";
}
exit;
```

The `for` statement could just as well have been used in the previous script, but the `foreach` keyword seems a little clearer in the context of going through the list of names. It also shows a very different loop construct than would be found in JavaScript. Each of the values in the variable is brought out almost like an array. However, as you can see, no array is declared.

The `while` loop in Perl is about the same as it is in JavaScript. The top of the loop contains the termination condition, and while that condition is not met, the loop iterates. The following is a simple generic example:

```
while ($counter < 10) {
      print "Give me some JavaScript<br>";
      $counter++;
      }
```

The `until` loop waits until a certain condition is met, looping until it is met. Note the differences between the following (`until`) and previous (`while`) script segments.

```
until ($counter == 10) {
      print " Give me some JavaScript<br>";
      $counter++;
      }
```

vFile Handling in Perl

One of Perl's attractions to designers is its capability to read and write text files. The attraction lies in the fact that designers can design their sites without having to set up a database (such as Access or MySQL) to store data. However, compared to PHP and ASP, in which the files are sophisticated relational databases, the files can be simple, plain-vanilla text files in a CGI page and Perl—the kind you can create with Notepad or SimpleText. To get started, create a simple text file using Notepad, SimpleText, or your favorite text editor. Begin by creating a list of names, separated by commas, and save them as a text file named customers.txt, as in the following list:

```
Lisa,Marge,Homer,Bart,Maggie,Ned,Mo
```

Place your text file on the server in the cgi-bin directory. Then, using CHMOD, initialize the file using 666 as the type. (With `CHMOD 666`, the owner, group, and everyone have read and write permissions, but none has the search/execute permission.) Next, type in the following script and save it as readData.pl:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
open(CUS,"customers.txt");     #Open the file using CUS to identify
the file
@cusList=<CUS>;                #Put the file contents into an array
print @cusList;
```

```
                        #Show on the screen what is in the file
close(CUS);                   #Close the file
exit;
```

After you have saved the file, put it into the same cgi-bin directory as the text file. Initialize readData.pl with `CHMOD 755`, and then open your browser and execute the script. You will see *exactly* what you put into the text file.

Now that you can see how easy it is to read and display the contents of a text file using Perl in a CGI page, you need to understand some very basic Perl statements and structures behind the process.

The open Function

The `open` function in Perl requires, at a minimum, only two arguments—a file reference name and the actual file's URL. The file reference name (filehandle) can be any name that you want, and it works like a nickname for the actual file. The basic format is the following:

```
open(fileHandle, "fileName.xxx");
```

When both the CGI page and the data file are in the cgi-bin directory, the URL is simply a filename, as shown in the example in the previous section. The file reference name or filehandle is conventionally capitalized; if you prefer, you can use the program's normal input channel name, STDIN. However, generally, you want to have a filehandle that is somehow associated with the file's contents or name.

When you open a file using Perl, you have three key options (among others):

- Read the file:

-
  ```
  Open(MYFILE, "fileName")
  ```
- Create a file and write to it:

-
  ```
  Open(MYFILE, ">fileName")
  ```
- Append an existing file:

-
  ```
  Open(MYFILE, ">>fileName")
  ```

Using *write* and *append*

Writing to a new file or appending an existing one is quite simple with Perl. Keeping in mind that most users will be appending existing files rather than creating new ones, the following two examples show the steps in first creating a new file and then appending the file. The sample file is a corporate one for the names and duties of personnel in an organization. Later, in using a single separator, a space, Perl will reformat the string for a clear output to the screen.

1. Create a text file and save it as posData.txt, but do not put anything in the file. Place the file into the cgi-bin directory, and initialize it with `CHMOD 666`.

2. Create a file to write data to the posData.txt file using the following script. (If you substitute names and positions in the $name or $job variable, be sure to keep any spaces out of the name or position. Use underscores [_] to connect the names or dashes [-] for the job title to create a string with no spaces.)

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
$name="Daniel_Gonzalez";
$job="programmer";
open(JOBS,">posData.txt");
#Include one space after $name and $job.
#The first print goes to the file, not the screen.
print JOBS "$name $job ";
close(JOBS);
#The second print goes to the screen.
print "Data added to file.";
exit;
```

   Save the file as write.pl, and put it into the cgi-bin directory. Initialize it with `CHMOD 755`. Test the script in your browser. After you have successfully launched the script, check the text file posData.txt. You should see this in the file:

```
Daniel_Gonzalez programmer
```

15. The append script is identical except for the values of the strings and the double arrow-bracket in the `open` function:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
$name="Nancy_Rogers";
$job="marketing-manager";
open(JOBS,">>posData.txt");
#Include one space after $name and $job.
#The first print goes to the file, not the screen.
print JOBS "$name $job ";
close(JOBS);
#The second print goes to the screen.
print "Data appended to file.";
exit;
```

   Save the script as append.pl, move it to the cgi-bin directory, and initialize it with `CHMOD 755`. Run the script in your browser. Again, open the posData.txt file in cgi-bin. This time you should see the following:

```
Daniel_Gonzalez programmer Nancy_Rogers marketing-manager
```

Having seen how to read files and write and append them using Perl and CGI, the next step is to read the files created with `write` and `append` and format the output. This last step is what Perl does well using very powerful pattern-recognition operators and functions. The next section examines Perl's regular expressions and pattern matching but, at this point, only the most simple formatting will be used, in an effort to clarify the process of formatting data with Perl.

Formatting with the *split()* Function

One of Perl's built-in functions, `split()`, can be used to divide up a string into multiple substrings. These substrings then are placed into an array and formatted for output. The general `split()` format used in the next example to cut up a string into array elements and place them into an array is as follows:

```
@arrayName = split(/character/,$stringName)
```

Basically, `split()` chops up the string along the lines that you specify in the pattern matching (character between the two slashes). Each array element is stripped of the matched character so that each element contains the substring between the matched characters. Because the data entered in the previous write and append examples used a space to separate both the name and the position, the names and positions are disconnected substrings. However, because the substrings (or elements in the array) contain names and positions in order of their connection to one another, by taking every other element in the array, it is possible to format and output them in a way that is connected. Try out the following script, and afterward go through the script analysis.

format.pl

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
open(POS,"posData.txt");
#Place the data into a scalar variable ($posn).
$posn=<POS>;
#Use the split function to divide up the parts by spaces (/ /) to be
placed in an array.
(@posnS).
@posnS=split(/ /,$posn);
#Step through every other element in array to group output by name
and position.
for($i=0;$i<=$#posnS;$i+=2) {
       #Replace underscores and dashes with spaces.
       $posnS[$i] =~ tr/_/ /;
       $posnS[$i+1] =~ tr/-/ /;
       print "$posnS[$i] position title is > $posnS[$i+1]<br>";
       }
close(POS);
exit;
```

Save the script, put the script file into the cgi-bin directory, and initialize it with `CHMOD 755`. When you run it from your browser, you will see this:

```
Daniel_Gonzalez position title is > programmer
Nancy_Rogers position title is > marketing-manager
```

As you append more names and positions, the output automatically will format it in this order and arrangement, adding new entries to the bottom of the list.

In looking at the script, the key points are in the lines that split the string and put it into an array, and the loop that pulls out the array elements and puts them on the screen. The first of these two key script lines is

```
@posnS=split(/ /,$posn);
```

The array `@posS` is made up of the scalar variable `$posn`, cut up into substrings at the split character. A blank space between the two slashes means that a blank space is the character where the string is cut up into substrings.

After the big string is broken into substrings and placed into an array, the next task is to pull each element out of the array and provide a format for output. In this case, a loop is used:

```
for($i=0;$i<=$#posnS;$i+=2) {
      print "$posnS[$i] position title is > $posnS[$i+1]<br>";
      $posnS[$i] =~  tr/_/ /;
      $posnS[$i+1] =~ tr/-/ /;
      }
```

The general way to find the length of an array minus 1 is to use this statement:

```
$#arrayName
```

To find the length of the array of `@posnS` for the termination condition in the `for` loop, use the statement `$#posnS`. The increment is set to 2 because the names are going to appear in only every other element. This expression does the trick for you:

```
$i += 2
```

Because the position follows the names, it is the next element value. Thus, the position is acquired by this expression:

```
$posnS[$i+1]
```

A line break tag `<br>` is placed after the position to create an orderly list.

Regular Expressions

The *sine qua non* of Perl is its regular expressions. Regular expressions are the codes used to specify a pattern that is to be matched with data. In the previous script used with the `split()` function, the regular expression operators `/ /` looked for a matching space in the data. All patterns in Perl express a pattern by placing the pattern between two slashes. For example, the following are all legitimate patterns in regular expressions:

```
/55/
/word/
/The stuff that dreams are made of/
```

The pattern matches if the string between the slashes matches a substring (a string segment) in the comparison data.

When you need to find several different matching strings, use the vertical bar ("pipe") character to separate the search elements. For example, the following has four patterns that can be matched with the data:

```
/honesty|integrity|persistence|wisdom/
```

Some Fundamental Operators in Regular Expressions

Perl has a number of different operators that can be used in regular expressions, but the most basic are the following:

- **=~** Contains (matches)
- **!~** Doesn't contain (doesn't match)

The matches work on a Boolean principle and are evaluated as `true` or `false`. As a result, you will find them used extensively in conditional statements. For example, the following conditional script looks to see whether a compound match exists and branches a certain way, depending on the outcome:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
$application="some experience low scores";
if ($application =~  /experience|high scores/) {
      print "You're hired!";
} else {
      print "We'll let you know.";
      }
exit;
```

The script compares the string `$application` with the match pattern. Because the word `experience` is contained in the string and the pattern, the expression evaluates as `true`.

You can also use regular expressions to substitute one substring for another. The format is

```
$stringName =~ s/oldString/newString/
```

The `oldString` pattern is sought in the string (`$stringName`) and, if found, the `newString` replaces the `oldString` substring. The following example illustrates one use of substitution:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
$thankYou="Thank you for your gift.";
$amount=200;
if($amount > 150) {
      $thankYou=~  s/gift./generous gift!/;
      }
print "$thankYou";
exit;
```

If you want to substitute one character for another, use this format:

```
tr/oldCharacter/newCharacter/
```

The only difference between using `tr///` (transliteration) and `s///` is that `tr///` scans a string one character at a time. However, you can put in ranges of characters, such as

```
tr/a-p/A-P/
```

Here, all lowercase characters in the range from `a` to `p` will be capitalized.

Perl also has pattern modifiers. The most important for designers might be the non–case-sensitive modifier `/i`. By placing the `/i` modifier at the end of a pattern, the letter cases will be ignored. For example, the following script excerpt would result in a match:

```
$bigshot = "HOT STUFF";
if($bigshot =~ /hot/i) {….
```

In most instances, Perl is *case-sensitive,* so, without having the capability to ignore cases in a match, you'd have a good deal more coding to make sure that a match was not ignored because of case.

Passing Data to CGI from HTML

In the previous sections in this chapter, you saw how to place data into a data file and how to pull it out and format it for the screen (format.pl). Using an HTML form and JavaScript, you can create a page that makes it easy to enter the data that you want and have it written to a text file stored in cgi-bin. You can also use JavaScript to help you preformat the data so that it will be easier to pull out and format using Perl.

Setting Up the Front End

The "front end" is where the user interfaces with the data. She enters the data, and it is passed to a CGI page and then written to a text file in a specific format. The interface also should have a means to read the data in the file. Thus, you will probably want to use frames. One frame will be used for data entry and providing buttons to read the data. The other frame is used for displaying data from the text file. A common external CSS file uses a French color pattern from Leslie Cabarga's *Designer's Guide to Global Color Combinations* (North Light Books, 2001).

filePass.css

```
h2 {
text-align:center;
       font-size:18pt;
       font-family:verdana;
       color:#e00021;
       background-color:#b8bab9;
       font-weight:bold
```

```
      }
#labels {
font-family:verdana;
      color:#b8bab9;
      background-color:983094;
      font-size:11pt;
      font-weight:bold
      }
a {
      font-family:verdana;
      color:#b8bab9;
      background-color:983094;
      font-size:11pt;
      font-weight:bold;
      text-decoration:none
      }
```

Next, you need to set up the frameset. With only two frames, the task is an easy one.

dataSet.html

```
<html>
<frameset cols="*,*" frameborder=0, border=0>
<frame name="dataIn" src="appendData.html" frameborder=0 border=0>
<frame name="dataOut" src="showData.html" frameborder=0 border=0>
</frameset>
</html>
```

The next segment of the front end is the core. It uses JavaScript to help format the data. Usually, some kind of verification would be included in the JavaScript as well, but, in this example, I wanted to focus on formatting text. The first JavaScript function simply adds an underscore character to link the first and last names. The second function looks for a space between any words in the window and places a dash between them, if one is found. Note how a regular expression, very much like the ones in Perl, is used in the second JavaScript function to add a dash (-) where any spaces exist. One of the two Perl scripts that accompanies this set called (format.pl) was used previously in this chapter. The JavaScript formatting sets up the data sent to the CGI page so that the format.pl script can be used again.

```
<html>
<head>
<link rel="stylesheet" href="filePass.css" type="text/css">
<title>Append Data to an existing data file </title>
<script language="JavaScript">
function addUS() {
      var firstForm=document.top;
      var secondForm=document.bottom;
      var fullName=firstForm.fName.value + "_" +
firstForm.lName.value + " ";
      secondForm.wName.value=fullName;
      }
function checkPos() {
      var t= document.top;
      var b=document.bottom;
      var spacer=t.position.value;
      //Use a regular expression to swap spaces for dashes
      var fixer= spacer.replace(/ /g,"-");
      b.positionF.value=fixer;
      }
//Put both functions together into a whole
function fixAll() {
addUS();
checkPos();
}
</script>
</head>
<body onLoad="document.forms[0].reset()" bgColor="#528490">
<h2>Fill in the form with your name and position.</h2>
 <div ID="labels">
<form name="top">
      Enter first and last names:<br>
      <input type=text name="fName">
      <input type=text name="lName"><br>
      Enter your position:<br>
      <input type=text name="position">
</form>
</div>
<form name="bottom" method=POST
action="http://www.sandlight.com/cgibin/
formAppend.pl"onSubmit="fixl  ()" target="dataOut">
      <input type=hidden name="wName">
      <input type=hidden name="positionF">
      <input type=submit>
</form><p>
<a href="http://www.sandlight.com/cgi-bin/format.pl"
target="dataOut">View Data
</a></p>
</body>
</html>
```

The two functions preprocess the data and store it in the second, hidden form. The data in the hidden form are sent to the CGI page. Before explaining what happens in the CGI page, the frameset needs to be completed with a "holding page" for the data that will appear in the frame later.

showData.html

```
<html>
<head>
<link rel="stylesheet" href="filePass.css" type="text/css">
</head>
<body bgcolor="#983094">
<h2>Data appears in this window:</h2>
</body>
</html>
```

That's all of the front end that has to be created. However, you must think of front ends and back ends always working together. So, the back end or server-side script in Perl must be matched to the type of data and method (`POST` or `GET`) used to send the data. This next section explains how things work on the CGI end of the equation.

Front-End and Back-End Connections: Interpreting Form Data in Perl

When you use a form to send data, it can be sent using the `POST` or `GET` methods. When data are sent using `GET`, this line places the data into a scalar variable:

```
$data=$ENV{'QUERY_STRING'};
```

If, as has been done in this example, your method is `POST`, you use this line to place data into a specified scalar variable:

```
read(STDIN, $data, $ENV{'CONTENT_LENGTH'});
```

(In this example, `$data` is used, but you can use any name you want.)

Because `POST` was used, the second of the two standard remote "data-catching" lines was employed in the script. The web sends a big hash called `%ENV` to CGI when a CGI script executes. All environmental variables are stored in the hash, and the line `$ENV{'CONTENT_LENGTH'}` specifies which environmental variable to use—in this case, `CONTENT_LENGTH`.

`STDIN` is a keyword in Perl for "standard input," and it specifies the standard input channel used to send data to a CGI script. So, basically, the line channels in data in the `POST` format and drops it into a scalar variable of your choice. Whether your form has 1 or 100 windows of data, all that Perl sees is one big string to make sense of. That's what will be stored in `$data` (or whatever name you use for a scalar variable).

Using the input line to extract data sent from an HTML form, the rest of the work for the Perl script is to format it using regular expressions for storage in a text file. This next script does exactly that:

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
read(STDIN, $fromPost, $ENV{'CONTENT_LENGTH'});
open(JOBS,">>posData.txt");
#Data from form is stripped of input names
$fromPost =~  s/wName=//;
$fromPost =~  s/\+&positionF=/ /;
$fromPost .= " ";
#Data is sent to text file using print statement
print JOBS "$fromPost";
close(JOBS);
#Let the viewer know the data are appended.
print "Data appended to file.";
exit;
```

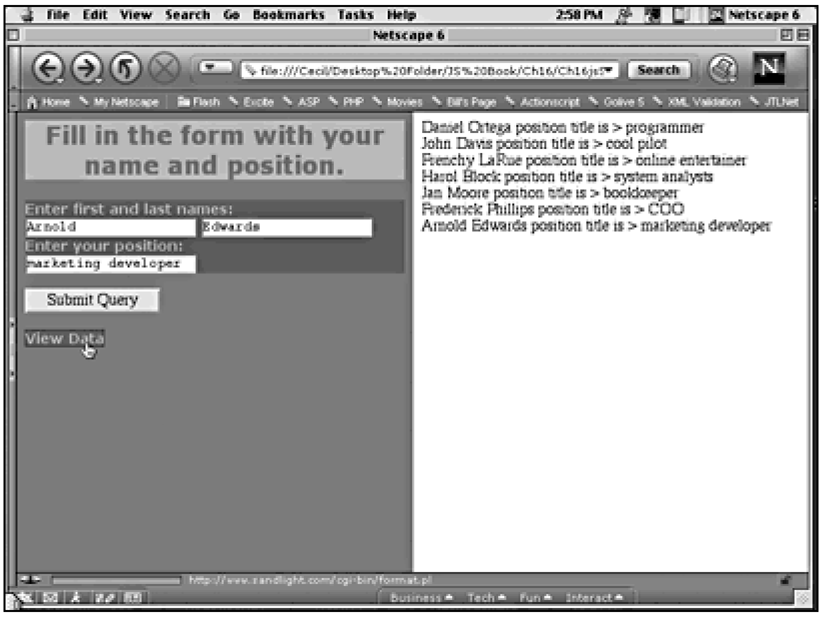
NOTE

As soon as you write to a file, it clears all the records from the file. Therefore, you want to have some kind of existing file to append data to. The easiest way is to *create an empty text file, save it in the cgi-bin directory, and use* `CHMOD 666` *to initialize it. Then the file can be appended indefinitely.*

Special characters are used to separate the form names when sent from the HTML form to the CGI page. Depending on the input form element names you used, you will be stripping different names than the ones shown previously. However, it is advisable to keep your form name short and clear when dealing with CGI pages, to make it easy on yourself when removing them from the data.

To finish up, all you need to do is to call on the script format.pl that was used in a previous section. (See the earlier section "Formatting with the `split()` Function.") It will read the data and present it in the right window of the frameset. Figure 16.2 shows the pages displayed in a browser window.

Figure 16.2. JavaScript, HTML, CGI, and Perl work well together.


File Edit View Search Go Bookmarks Tasks Help
2:58 PM
Netscape 6
file:///Cecil/Desktop%20Folder/JS%20Book/Ch16/Ch16js
Search
Home My Netscape Flash Excite ASP PHP Movies Bill's Page Actionscript Golive 5 XML Validation JTLNet
Fill in the form with your name and position.
Enter first and last names:
Enter your position:
Submit Query
View Data
http://www.sandlight.com/cgi-bin/format.pl
Business Tech Fun Interact

Arnold | Edwards

marketing developer

```
Daniel Ortega position title is > programmer
John Davis position title is > cool pilot
Frenchy LaRue position title is > online entertainer
Harol Block position title is > system analysts
Jan Moore position title is > bookkeeper
Frederick Phillips position title is > COO
Arnold Edwards position title is > marketing developer
```

Summary

JavaScript plays a secondary role between forms and CGI pages, but it can be an excellent preprocessor for use with CGI scripts. Besides being used for verification, JavaScript can be used for preformatting data to be sent to a CGI script for placement into data files. The preformatting can make it easier to create a script that will store and retrieve the data for clearly formatted display.

Perl is a very rich language, and this chapter has just scratched the surface of working with CGI and Perl. Several good books on Perl and CGI are available, and the third edition of *Programming Perl* (O'Reilly, 2000) is a thorough resource for learning much more about Perl and its relationship to CGI and HTML. The wide range of regular expressions in Perl helps to understand and fully use regular expressions in JavaScript as well.

Chapter 17. Working with XML and JavaScript

CONTENTS>>

The XML Mystique

The Extensible Markup Language (XML) is one of those languages that you hear a lot about, and generally in the superlative, but not too many people are exactly sure what it is. At this point in time, both Netscape Navigator and Internet Explorer are on the verge of fully connecting JavaScript and XML using the W3C Document Object Model (DOM). Because the HTML, JavaScript, and XML DOMs are beginning to form around the same object model, you can better understand where JavaScript and HTML are headed by understanding XML.

Microsoft has provided one way of examining XML documents with IE5+ using platform-specific keywords on Windows platforms. As both NN6 and IE6 mature, working with XML will not require a separate module to load XML. So, even though limited to the Windows platform and IE5+ browser, you can see how JavaScript can be used to pull data out of an XML file and display it on the screen.

If you have ever seen stockbrokers at work on Wall Street, you might have noticed that they have several computers and monitors. What you are seeing is actually different databases being sent over different proprietary systems. Instead of needing different systems for each database, XML can put *any* database into a format that can be read by any computer with the right browser. At this point in time, XML is ahead of the browsers.

Because this single chapter is a scratch on the surface of XML, I highly recommend a more thorough treatment of the topic. *Inside XML,* by Steven Holzner (New Riders, 2001), is an excellent source of XML and has a great chapter on using JavaScript with XML. Mr. Holzner's book has more than 1000 pages that look into just about every nook and cranny of XML, and it is well worth taking a look at.

What Is XML?

XML organizes and structures data for the web. In many ways, it is like a database; in others, it is like a text file storing data. However, XML looks a lot like an HTML page as well, but with no built-in formatting tags. XML tags only order data. All of the tag names in XML are ones provided by the designer. For most XML pages, you can determine approximately what the structure is by examining the file. The following page is an example:

```
<?xml version="1.0" ?>
<writers>
     <pen>
            <name>Jane Austin</name>
     </pen>
     <pen>
           <name>Rex Stout</name>
     </pen>
</writers>
```

You can write this document in your favorite text editor, such as Notepad in Windows or SimpleText on the Macintosh. Save it as writers.xml. (All XML documents can be written and saved as text files.) If you load the XML page into IE5+ or NN6+, you will see this:

```
Jane Austin Rex Stout
```

XML is for structuring data, not formatting it, and you need something to show that data in a useful way. Most developers use Cascading Style Sheets (CSS). For example, the following CSS script provides formatting in the form of an 11-point bold navy Verdana font for the data in the XML file:

```
name {
      display:block;
      font-size: 14pt;
      color: navy;
      font-weight: bold
      }
```

By saving the file as an external style sheet named scribe.css, you can use it to format the elements with the tag label `name`. Note that `name` is *not* a dot-defined class or an ID. It is the name of the label in the XML script.

XMLsee.xml

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/css" href="scribe.css" ?>
<writers>
     <pen>
            <name>Jane Austin</name>
     </pen>
     <pen>
            <name>Rex Stout</name>
     </pen>
</writers>
```

The output is now formatted, and your screen shows this:

```
Jane Austin
Red Stout
```

You can use the same style sheet with your HTML/JavaScript pages as you do with XML. However, in creating the style sheet, this line in the CSS script has the effect of blocking the text on separate lines:

```
display: block;
```

The Rules of Writing XML

XML is a markup language that uses tags, like HTML, but you will find many differences as well. The following list shows what you must be aware of in creating XML files:

- The user defines the element names. (An opening and closing tag constitute an element: `<pen>…</pen>`.) XML has no formatting tags of its own.
- *Internet Explorer 5+* and *Netscape Navigator 6+* are required for displaying XML in a browser. Older browsers cannot display XML.
- Like JavaScript, XML is case-sensitive. The tag `<Pen>` is not the same as `<pen>`.
- All XML elements are containers and must have closing tags (for example, `<pen>` and `</pen>`). You could not have `<p>` without `</p>`, as with HTML.
- Each XML element requires a document type definition (DTD) or schema to be well formed.
- An element in XML is a self-contained mini-XML document.

Declaring an XML Document

To create an XML document, you need to first declare the document as an XML document. You do so with the following line:

```
<?xml version="1.0" ?>
```

You can add more information for different languages, but for the purposes at hand, just start off your XML scripts with this single line.

The Root Element

Following the XML document declaration, you need a root declaration. The *root element* encompasses everything that you put into the XML document. The other elements must be between the tags identifying the beginning and end of your root element. In the XML example that we've been using, the root element is `<writers>`. (Note that the comment tags are the same as those used in HTML.)

```
<?xml version="1.0" ?>
<!--First the root element -->
<writers>
<!--Rest of the tags between the opening and closing root tags -->
</writers>
```

The root element is important because it is a reference point used by JavaScript to identify the hierarchy that leads to different child elements.

Filling in the Root

The parent-child relationship in XML is one of containers. The root element contains child elements. If one of the root's child elements contains a container, it is the parent of the contained tags yet still the child of the root element.

```
<root>      Parent to all elements
      <child>      Child of <root> Parent of <grandchild> elements
               <grandchild>Dashiell Hammett</grandchild> Child of
<child>
               <grandchild>Toni Morrison</grandchild> Child of
<child>
      </child>
</root>
```

The `<root>` element is the parent of all, and the `<child>` element is the child of the root element and parent of both of the `<grandchild>` elements. The two `<grandchild>` elements are siblings. In the JavaScript DOM objects, you will see methods referring to child, parent, and sibling; these methods address the set of parent and child elements references.

Reading and Showing XML Data with JavaScript

As noted previously, Version 6 JavaScript browsers seem to be coming together over the W3C DOM. Several key methods and properties in JavaScript can help in getting information from an XML file. In the section, a very simple XML file is used to demonstrate pulling data from XML into an HTML page using JavaScript to parse (interpret) the XML file. Unfortunately, the examples are limited to using IE5+ on Windows. (The same programs that worked fine using IE5+ on Windows bombed using IE5+ on the Mac using either OS 9+ or OS X.)

However, the great majority of keywords used in the scripts are W3C DOM–compliant, and the only keywords required from the Microsoft-unique set are `XMLdocument` and `document.all()`. All of the other keywords are found in NN6+. Table 17.1 shows the W3C JavaScript keywords used in relationship to the XML file examples.

Table 17.1. Selected Element Keywords in JavaScript

Property	Meaning
`documentElement`	Returns the root element of the document
`firstChild`	Is the first element within another element (the first child of the current node)
`lastChild`	Is the last element within another element (the last child of the current node)
`nextSibling`	Is the next element in the same nested level as the current one
`previousSibling`	Is the previous element in the same nested level as the current one
`nodeValue`	Is the value of a document element
Method	**Meaning**
`getElementsByTagName`	Used to place all elements into an object

Finding Children

To see how to pull data from an XML file, all examples use the following XML file. The intentional simplicity of the XML file is to help clarify using JavaScript with

XML and does not represent a sophisticated example of storing data in XML format.

writers.xml

```
<?xml version="1.0" ?>
<writers>
     <EnglishLanguage>
               <fiction>
                       <pen>
                             <name>Jane Austin</name>
                             <name>Rex Stout</name>
                             <name>Dashiell Hammett</name>
                       </pen>
               </fiction>
     </EnglishLanguage>
</writers>
```

The XML file contains a typical arrangement of data using a level of categories that you might find in a bookstore or library arrangement. It is meant to be intuitively clear, as is all XML.

The trick in all of the following scripts is to understand how to find exactly what you want. The first three scripts that follow use slightly different functions to find the first child, last child, and sibling elements. The first script provides the entire listing, and the second two just show the key JavaScript function within the script. They all use the following common CSS file.

readXML.css

```
body {
      font-family:verdana;
      color:#ff4d00;
      font-size:14pt;
      font-weight:bold;
      background-color:#678395;
}
div {background-color:#c1d4cc;}
#blueBack {background-color:#c1d4cc}
```

To read the first child of an element, the reference is to `document.firstChild`. Given the simplicity of the sample XML file (writers.xml), the script just keeps adding . `firstChild` to each of the elements as it makes its way to the place in the XML file where the information with the data can be found.

However, before even going after the first child of the `<name>` element, the HTML page sets up a connection to the XML page using an `<xml>` container understood by *Internet Explorer 5+* in a Windows context. (At the time of this writing, IE6 was available, and it worked fine with the following scripts, but only on a Windows PC.) The ID `writersXML` is defined as the XML object first, and then it becomes part of a document, myXML, in this line:

```
myXML= document.all("writersXML").XMLDocument
```

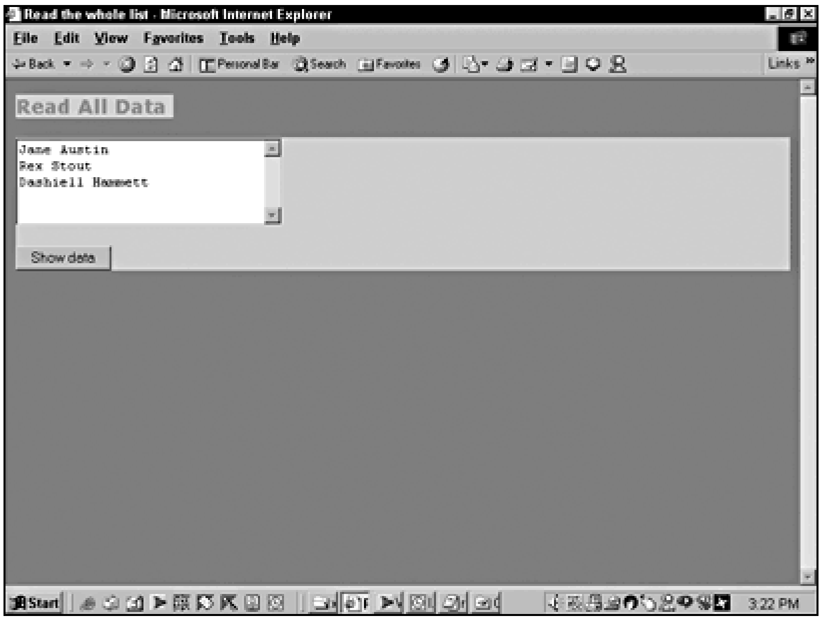
The `document.all().XMLDocument` is a Microsoft IE subset of JavaScript. After this point, though, the JavaScript is pure W3C DOM and is consistent with NN6+.

With this line, `writersNode` is defined as the root element of the XML file with the `documentElement` property:

```
writersNode = myXML.documentElement
```

Its first child is the `<EnglishLanguage>` node, so the variable `languageNode` is defined as `writersNode.firstChild`. Then the rest of the nodes in the XML document are defined until the first child of the `<name>` node is encountered and its node value is placed into a variable to be displayed in a text window. All of the processes are placed into the `findWriter()` user function. Figure 17.1 shows how the page looks when opened in a browser.

Figure 17.1. The first child of `<pen>` is displayed.


Read the whole list - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Read All Data
Show data

```
Jane Austin
Rex Stout
Dashiell Hammett
```

readFirstChild.html

```
<html>
<head>
<link rel="stylesheet" href="readXML.css" type="text/css">
<title>Read First Child</title>
<xml ID="writersXML" SRC="writers.xml"></xml>
<script language="JavaScript">
function findWriter() {
      var myXML, writersNode, languageNode,
      var penNode,nameNode,display
      myXML= document.all("writersXML").XMLDocument
      writersNode = myXML.documentElement
      languageNode = writersNode.firstChild
      fictionNode = languageNode.firstChild
      penNode = fictionNode.firstChild
      nameNode = penNode.firstChild
      display =nameNode.firstChild.nodeValue;
      document.show.me.value=display
      }
</script>
</head>
<body>
<span ID="blueBack">Read firstChild</span>
```

```
<div>
<form name="show">
<input type=text name="me">
<input type="button" value="Display Writer" onClick="findWriter()">
</form>
</div>
</body>
</html>
```

Reading the last child uses an almost identical function. However, when the script comes to the parent element `<pen>` of the `<name>` node, it asks for the last child, or simply the one at the end of the list before the `</pen>` closing tag.

readLastChild.html (Function Only)

```
function findWriter() {
      var myXML, writersNode, languageNode,
      var penNode,nameNode,display
      myXML= document.all("writersXML").XMLDocument
      writersNode = myXML.documentElement
      languageNode = writersNode.firstChild
      fictionNode = languageNode.firstChild
      penNode = fictionNode.firstChild
      nameNode = penNode.lastChild //Here is the key line
      display =nameNode.firstChild.nodeValue;
      document.show.me.value=display
      }
```

Because the DOM contains keywords for the first and last children, finding the beginning and end of an XML file is pretty simple. What about all of the data in between? To display the middle children, first you have to find the parent and start looking at the next or previous sibling until you find what you want. This next function shows how that is done using the `nextSibling` property.

readSibling.html (Function Only)

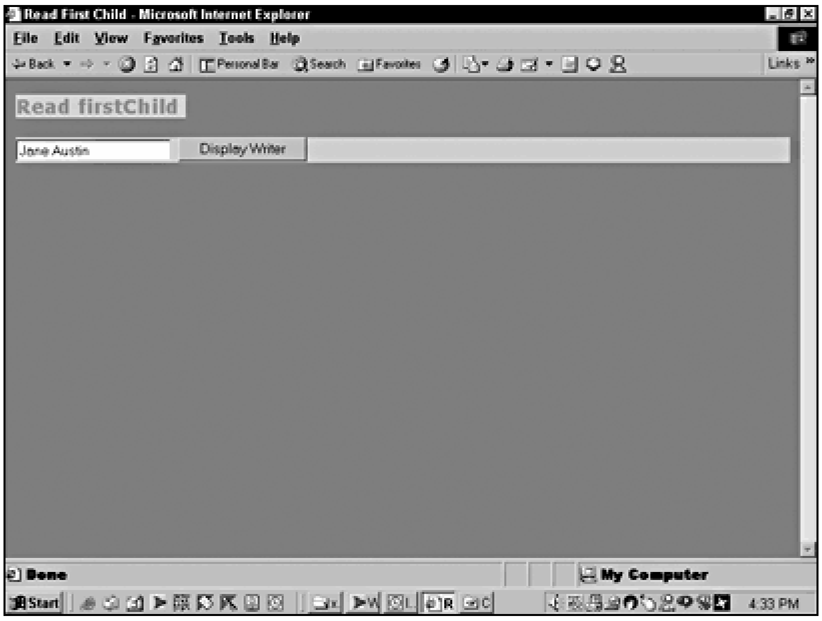
```
function findWriter() {
      var myXML, writersNode, languageNode
      var penNode,nameNode,nextName,display
      myXML= document.all("writersXML").XMLDocument
      writersNode = myXML.documentElement
      languageNode = writersNode.firstChild
      fictionNode = languageNode.firstChild
      penNode = fictionNode.firstChild
      nameNode = penNode.firstChild
      nextName=nameNode.nextSibling //Not the first but the next!
      //The first child is the only child in the next node.
      display =nextName.firstChild.nodeValue;
      document.show.me.value=display
      }
```

The three functions differ little in what they do or how they do it. However, using this method to find a single name in a big XML file could take a lot of work. As you might have surmised, because the XML file is part of an object, you can extract it in an array-like fashion.

Reading Tag Names

Instead of tracing the XML tree through child and parent nodes, you can use the `getElementByTagName()` method. By specifying the tag name that you're seeking, you can put all of the tag's values into an object and pull them out using the `document.item()` method. The process is much easier than going after first and last children or siblings and, I believe, much more effective for setting up matching components. The following script is similar to the others and uses the same external Cascading Style Sheet. The form is slightly different at the bottom, so the whole program is listed rather than just the function. Figure 17.2 shows the output in the browser.

Figure 17.2. All of the data in the specified tag category are brought to the screen.


Read First Child - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Read firstChild
Jane Austin
Display Writer
Done
My Computer
4:33 PM

readNode.html

```
<html>
<head>
<link rel="stylesheet" href="readXML.css" type="text/css">
<title>
Read the whole list
</title>
<xml ID="writersXML" SRC="writers.xml"></xml>
<script language="JavaScript">
function findWriters() {
      var myXML, myNodes;
      var display="";
      myXML= document.all("writersXML").XMLDocument;
      //Put the <name> element into an object.
      myNodes=myXML.getElementsByTagName("name");
      //Extract the different values using a loop.
      for(var counter=0;counter<myNodes.length;counter++) {
            display += myNodes.item(counter).firstChild.nodeValue +
"\n";
      }
      document.show.me.value=display;
}
```

```
</script>
</head>
<body>
<span ID="blueBack">
Read All Data
</span>
<div>
<form name="show">
<textarea name="me" cols=30 rows=5></textarea><p>
<input type="button" value="Show all" onClick="findWriters()">
</form></div>
</body>
</html>
```

At this stage in browser development, the great majority of terms used in extracting data from an XML file are cross-browser–compatible, especially when Version 6 of both browsers are compared side to side. In large measure, this is due to the fact that the browser manufacturers are beginning to comply with the W3C DOM recommendations. The Microsoft extensions to the W3C DOM could become adopted as part of the DOM (as some have already), or the W3C DOM could develop functional equivalents. However, at the time of this writing, there might not actually be a W3C DOM–compliant method of the crucial first step of loading an XML document into an HTML page. So, in the meantime, which I hope is short, it is necessary to use the single-browser, single-platform techniques shown previously.

Well-Formed XML Pages

A well-formed XML page requires either a DTD or a schema (exclusively Microsoft).The DTD tells the parser what kind of data is contained in the XML file. If XML pages were parsed only by JavaScript, no one would worry too much about DTD. However, when a browser parses an XML file, it looks at the DTD to determine what kind of data are in the file and how it is ordered. XML validators scan XML files and determine whether they are valid, but browsers do not validate XML files. (A good validator can be found at Brown University's site, www.stg.brown.edu/service/xmlvalid/.) If an XML file is not valid, problems are likely to crop up.

Validation takes a little extra work, but you will know that your XML file is well formed, and it won't run into problems down the line somewhere. Using the example XML file used previously, a DTD has been added in the following file, writersWF.xml.

All document type definitions begin with this line:

```
<!DOCTYPE rootName [
```

Because `writers` is the root element, it goes in as the root name. Next, the first child of the root is declared—in this case, the child is `<EnglishLanguage>`, so the `!ELEMENT` declaration is as follows:

```
<!ELEMENT writers (EnglishLanguage)>
```

You continue with !ELEMENT declarations until all of them are made. If more than one instance of an element is within another element's container, a plus sign (+) is added to the end of the element name. Because three nodes using <name> are within the <pen> element, the !ELEMENT declaration for <name> has a plus after it:

```
<!ELEMENT pen (name+)>
```

Finally, close up the !DOCTYPE declaration using this code:

```
]>
```

Your file is ready for validation. The complete listing follows.

writersWF.xml

```
<?xml version="1.0" ?>
<!DOCTYPE writers [
<!ELEMENT writers (EnglishLanguage)>
<!ELEMENT EnglishLanguage (fiction)>
<!ELEMENT fiction (pen)>
<!ELEMENT pen (name+)>
<!ELEMENT name (#PCDATA)>
]>
<writers>
     <EnglishLanguage>
             <fiction>
                   <pen>
                         <name>Jane Austin</name>
                         <name>Rex Stout</name>
                         <name>Dashiell Hammett</name>
                   </pen>
              </fiction>
     </EnglishLanguage>
</writers>
```

Will this new validated file work with the example scripts provided previously? You bet! In all of the previous files showing how JavaScript parses XML files, substitute writersWF.xml for the original writers.xml in this line:

```
<xml ID="writersXML" SRC="writers.xml"></xml>
```

When you re-run the script in IE5+ on your Windows PC, you will see exactly the same results. The only difference is that now your XML file is well formed.

XHTML

Using XML, HTML, and JavaScript together can be a bit confusing. You might want to take a look at XHTML, where you will find better integration between XML and HTML. XHTML brings well-formed code to HTML. At the same time, you can insert JavaScript into the middle of XHTML pages for adding dynamic action. A good place to start is with *XHTML,* by Chelsea Valentine and Chris Minnick (New Riders, 2001).

Summary

To say that this chapter just scratched the surface of XML is an understatement. However, understanding even a little of how JavaScript and XML work together is a preview of the direction of the W3C DOM and the future of JavaScript. The capability to pull data out of an XML file is a bit easier than pulling it out of a database using PHP, ASP, or CGI as intermediaries. Using server-side JavaScript and a well-formed XML file, you can perform just about anything that you can with files stored in more traditional databases. (Getting data into an XML file and storing it, though, is a horse of a different color.)

At the point of this writing, all of JavaScript and the W3C DOM are on the verge of providing a robust language for manipulating data stored in XML files. Designers are encouraged to follow the changes and to see how XML can be used effectively for their clients, and many clients are now demanding the XML structures for their data. Taking some time to learn more about XML is essential for keeping up with all of the changes taking place on the web, especially in storing and retrieving structured data.

Chapter 18. Flash ActionScript and JavaScript

CONTENTS>>

One of the most popular applications for creating interactive web sites is Flash. Flash SWF files reside in web pages, and these files can communicate with JavaScript. Flash pages are very easy to use for creating sophisticated animation at very low bandwidth. Both of the major browsers are shipped with Flash plug-ins, so cross-browser compatibility is not the issue that it is with JavaScript.

However, on many occasions, you will find that firing a JavaScript function from an HTML page with a Flash file can give you the design you want that cannot be accomplished by Flash alone. For example, one client wanted to have an external link to pages that showed some awards that her company had received. The pages with the awards were nice to see, but they really did not fit into the Flash design. Creating a JavaScript function that opened the pages in a separate HTML window made the pages accessible without requiring them to be embedded in the HTML page with the SWF file. Likewise, other uses can be discovered for JavaScript in a Flash environment. This chapter examines how the power of Flash and ActionScript (Flash's built-in language) can be enhanced by JavaScript, and vice versa.

ActionScript and JavaScript

To get the most out of this chapter, you will need to know *Flash 5* and something about its built-in language, ActionScript. Fortunately, ActionScript is almost identical to JavaScript, especially the newer versions of JavaScript. The main difference between the two is that ActionScript was designed to work with Flash's timeline and movie clip environment, and JavaScript was designed to work in an

HTML environment. The dot syntax is very similar, and most lines require a semicolon at the end. ActionScript follows the same semicolon placement rules that JavaScript does, except that the semicolons are *mandatory* in ActionScript and optional in JavaScript.

If you are wholly unfamiliar with ActionScript but know Flash, take a look at *ActionScript f/x and Design* (Coriolis, 2000), or dust off the ActionScript manual that accompanies *Flash 5* . In this chapter, only selected ActionScript elements are explained in any detail; while every attempt is made to provide enough explanation of ActionScript to see how a certain script accomplishes a goal, a little background will help.

Firing a JavaScript Function from Flash

Probably the most valuable and simple technique that you can use with JavaScript and Flash is calling a JavaScript function using this ActionScript format:

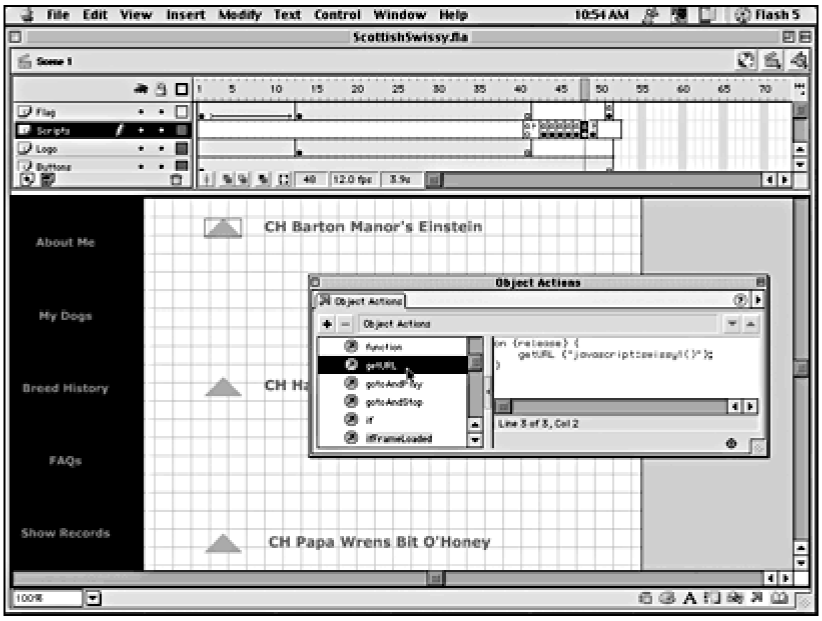
```
getURL("javascript:jsFunction()");
```

For example, in one project I wanted to call up HTML pages with lots of text. While text is relatively "light" in HTML, it can really bog down a Flash movie. I created three buttons in Flash, and each one contained this script, with a variation on the function `swissy1()-swissy2()` and `swissy3()`.

```
on (release) {
        getURL ("javascript:swissy1()");
}
```

Figure 18.1 shows the script in the Object Actions window being developed in Flash.

Figure 18.1. Linking JavaScript functions to a Flash movie is simple.



```
on (release) {
    getURL ("javascript:swissy1()");
}
```

Line 3 of 3, Col 2

After the HTML page was published from Flash, all of the functions were added to the HTML page that Flash generated. The following shows the revised script:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>ScottishSwissy</title>
<script language="JavaScript">
//This section was added
function swissy1(){
      open("swissy1.html","s1","scrollbars=1, width=500, height=400,
      resizable=yes")
      }
function swissy2(){
      open("swissy2.html","s2","scrollbars=1, width=500, height=400,
      resizable=yes")
       }
function swissy3(){
open("swissy3.html","s3","scrollbars=1, width=500, height=400,
resizable=yes")
}
</script>
</head>
<body bgcolor="#ffffff">
<a href="javascript:swissy1%28%29"></a>
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swf
lash.
cab#version=5,0,0,0" width="100%" height="100%">
<param name="movie" value="ScottishSwissy.swf">
<param name="quality" value="high">
<param name="bgcolor" value="#FFFFFF">
<embed src="ScottishSwissy.swf" quality="high" bgcolor="#FFFFFF"
width="100%"
height="100%" type="application/x-shockwave-flash"
pluginspage="http://www.macromedia.com/shockwave/download/index.cgi?P
1_Prod_Version
=ShockwaveFlash">
</object>
</body>
</html>
```

NOTE

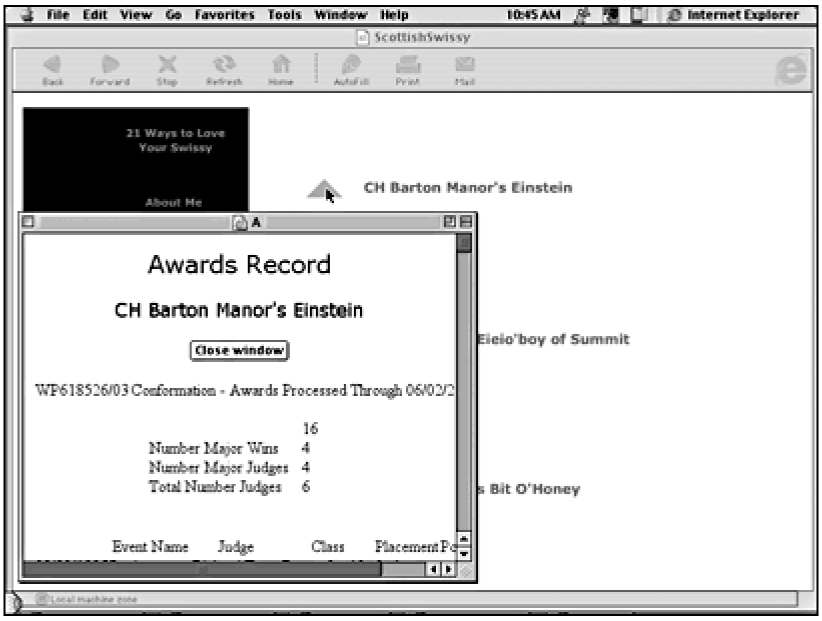
The previous script had some very long comments generated by Flash when it was published. They were not informative to understanding the connection between a JavaScript function and Flash, and they were removed.

Each of the HTML pages that appear contained the following JavaScript fired by a button to get the HTML page off the screen after it was viewed.

```
function closeIt() {
     window.close();
     }
```

Whenever an external HTML page is opened in a separate Window, be sure to include a Close button of some sort along with the appropriate script. Figure 18.2 shows the Flash page and external HTML page when working together.

Figure 18.2. New windows opened from Flash should always have a Close button.


File Edit View Go Favorites Tools Window Help
10:45 AM
Internet Explorer
ScottishSwissy
Back Forward Stop Refresh Home AutoFill Print Mail
21 Ways to Love Your Swissy
About Me
CH Barton Manor's Einstein
Eicio'boy of Summit
s Bit O'Honey
A
Close window
Local machine zone

Awards Record

CH Barton Manor's Einstein

WP618526/03 Conformation - Awards Processed Through 06/02/2

	16
Number Major Wins	4
Number Major Judges	4
Total Number Judges	6

Event Name Judge Class Placement Po

Passing Variables from Flash 5 to JavaScript

When the `fscommand()` statement was added to the Flash ActionScript lexicon way back in *Flash 3* , many developers were optimistic about easily passing variables from Flash to an HTML page via JavaScript. However, something happened along the way, and about the only browser around that supports the `fscommand()` and JavaScript is version 4 of Netscape Navigator. Tested on NN6.1, the `fscommand()` was not recognized in the browser, and while IE can handle a VBScript version of an `fscommand()` statement, it doesn't respond to the JavaScript version.

In Flash, the `fscommand()` has this format:

```
fscommand(command,args);
```

The two arguments in the function, `command` and `args`, can be two strings, variables, or a function and its arguments, as implied by the terms. However, `command` and `args`, the default terms in the normal mode of entering ActionScript, can be replaced by string literals or any variables that you want to include. Whatever you put in the `command` and `args` arguments in the `fscommand` action is passed to a JavaScript program that you create in the HTML page that contains the SWF files with the `fscommand` action. For example, in a button script, a developer might enter this:

```
On(release) {
      fscommand("Hope","Glory");
}
```

The two strings "Hope" and "Glory" are passed to two arguments in a JavaScript function. The function name is automatically generated by Flash, and all you have to do is to fill in the statements within the JavaScript function's curly braces. For example, if the previous fscommand action were used in a Flash movie and the movie were saved as hopeful, the JavaScript function generated would look like the following script segment in the Flash-generated HTML:

```
...
<SCRIPT LANGUAGE=JavaScript>
<!--
var InternetExplorer = navigator.appName.indexOf("Microsoft") != -1;
// Handle all the FSCommand messages in a Flash movie
function hopeful_DoFSCommand(command, args) {
     var hopefulObj = InternetExplorer ? hopeful : document.hopeful;
   //
   // Place your code here…
   //
}….
```

The function name, hopeful_DoFSCommand(command,args), is automatically generated by Flash when the movie is published and the HTML publishing setting template is set to Flash with FSCommand in Publishing Preferences. The extension attached to the Flash filename with an underscore, _DoFSCommand, links the function to the Flash fscommand action that sends the variables to JavaScript. (The line beginning var hopefulObj= is a conditional statement to jump to another section of the script that handles VBScript for Internet Explorer.)

The <EMBED> tag must include the name of the Flash movie and must set the swLiveConnect to true, as shown in the following tag segment:

```
<EMBED ..... swLiveConnect=true NAME=hopeful ...>
```

If your Publish settings are correct, all of this will be done for you.

Data Entered in Flash and Passed to JavaScript

To see how the process works, this next movie lets the user enter data in the Flash section of the movie and pass it to a form in HTML via JavaScript. The following steps guide you through the process:

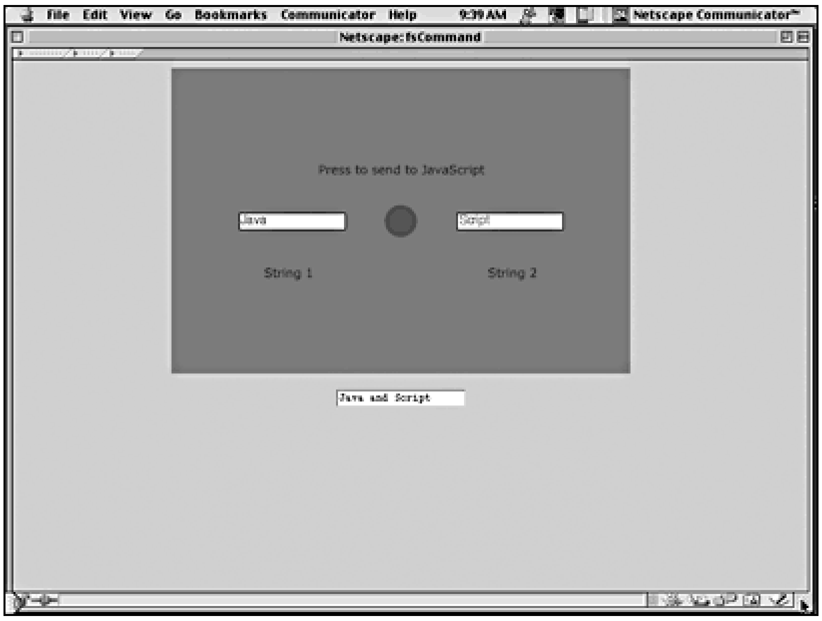
1. Open a new Flash movie and set the stage size to 450 by 300. Select Modify, Movie from the menu bar. This size movie will give you room at the bottom for an HTML form that will show the data passed from Flash.

2. Add layers for a total of four layers, with the names from top to bottom. Send to JavaScript, Labels, Data Entry, and Background.

3. Create a color in the Mixer panel with the values R=232, G=49, B=0 (red). Add that color to your Swatches panel. From the menu bar, select Modify, Movie; in the background color well, select the red that you just added to the Swatches panel.

4. Select the Data Entry layer and add two input text fields in the middle of the stage. (Use Figure 18.4 as a guide.) Select the left text field and, in the Variable text window in the Text Panel, type in **alpha**; for the text window on the right, use the variable name **omega.** Use a 12-point dark-color Verdana font. Lock the layer.

Figure 18.4. Data from Flash can be passed through JavaScript and put into an HTML form or variable.



5. Select the Labels layer and type in **String 1** using Static text under the left input window and **String 2** under the right one. Above both windows, type in **Press to send to JavaScript.** Use Figure 18.4 as a guide. Lock the layer.

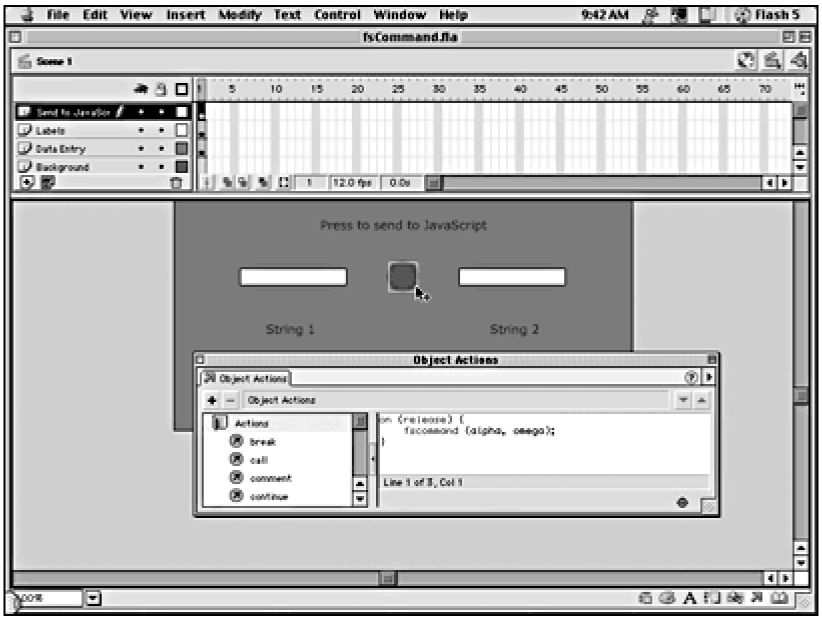
6. Select the Send to JavaScript layer. Add two colors to your swatches: R=0, G=91, B=0 (green) and R=141, G=43, B=78 (deep purple). Using the green for the stroke (3-point stroke) and purple for the fill, draw a circle with a 28pixel diameter. Select the circle and press the F8 key to open the Symbol Properties dialog box. Choose Button as the behavior, and name the button Frank or any other name that you want; then click OK. Place the button between the two input text fields.

7. Select the button and open the Object Actions panel, and insert the following script:

```

on (release) {
        FSCOMMAND (ALPHA, OMEGA);
}
```

Figure 18.3 shows the entry in Flash's Object Actions panel.

Figure 18.3. Entering the fscommand action in Flash.



```
on (release) {
    fscommand (alpha, omega);
}
```

8. Save the FLA file as fsCommand.fla. From the menu bar, select File, Publish Settings, HTML, Template, Flash with FSCommand. Click the Publish button on the right side of the HTML tab.

9. Open the HTML page created by Flash (fcCommand.html) in your text editor of preference (such as Notepad or SimpleText). Edit the page so that it conforms to the following script:

```

<HTML>
<HEAD>
<TITLE>fsCommand</TITLE>
<SCRIPT LANGUAGE=JavaScript>
function fsCommand_DoFSCommand(alpha,omega) {
      document.show.me.value=alpha + " and " + omega;
      }
</script>
</HEAD>
<BODY bgcolor="pink">
<center>
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#versi
on=5,0,0,0" ID=fsCommand WIDTH=450 HEIGHT=300>
<PARAM NAME=movie VALUE="fsCommand.swf">
<PARAM NAME=quality VALUE=high>
<PARAM NAME=bgcolor VALUE=#E83100>
<EMBED src="fsCommand.swf" quality=high bgcolor=#E83100 WIDTH=450 HEIGHT=300
swLiveConnect=true NAME=fsCommand TYPE="application/x-shockwave-flash"
PLUGINSPAGE="http://www.macromedia.com/shockwave/download/index.cgi?P1_Prod_Versi
on=ShockwaveFlash"></EMBED>
</OBJECT>
<form name="show">
<input type=text name="me">
</form>
</center>
</BODY>
```

```
        </HTML>
```

As you can see from the script, the JavaScript function has been moved to the head area of the script, a form has been added, and all of the extraneous material for working with VBScript has been eliminated. Otherwise, not a lot was changed from the original script generated by Flash. (If you so desire, you can leave the VBScript material in.) The argument names are `alpha` and `omega` instead of the generated `command` and `args`. (They could have been named `Bonnie` and `Clyde`, or anything else, for that matter. As long as the argument names match the terms used in the statement in the function, the correct data will be passed from Flash to JavaScript.)

When you load an HTML page into Netscape Navigator 4.x, you will see the page shown in Figure 18.4. By adding text in the Flash text windows labeled String 1 and String 2 and then pressing the button, the two strings will be placed in the HTML form at the bottom of the page with the word *and* concatenated between them.

The point of this exercise is to see how variables can be passed from Flash to an HTML page using JavaScript. Unfortunately, only *Netscape Navigator 4.x* supports using JavaScript in this way, and the newer versions of Navigator do not. In future versions of Flash and the major browsers, we hope that the functionality of the `fscommand` action is the main way that Flash data is passed to HTML. Between calling JavaScript functions from HTML and passing Flash data to HTML with `fscommand`, a complete cycle of data is possible using JavaScript.

Summary

Working with Flash is an important role for JavaScript. Passing data, variables, functions, and other information between an SWF file on an HTML page and the HTML page itself enables data from multiple sources to be used interchangeably. Having the capability to call JavaScript functions from Flash using the `getURL("javascript:jsfunction()")` action enables Flash to have most of the power that JavaScript has in working with HTML pages.

Unfortunately, limited exchange between Flash and JavaScript, even with the `fscommand` action, does not allow the kind of full exchanges between the SWF file and JavaScript as would be desired. Currently, only *Netscape Navigator 4.x* provides a platform on which JavaScript can receive and use data from Flash. However, in future editions of either Flash or the browsers, we hope for a functionally similar action.

In the meantime, keep in mind that JavaScript functions can be called from Flash. Everything from pop-up alert boxes to external windows can easily be passed from a JavaScript function to a Flash movie simply by a little ActionScript that gets some needed help from JavaScript.

Chapter 19. JavaScript and Other Languages

CONTENTS>>

This chapter explores the relationship between JavaScript and other languages that you might encounter or want to learn more about. Each of these applications has its own unique relationship to JavaScript, and the role of JavaScript is an adjunct or facilitating one. None of them relies fundamentally on JavaScript, but they can be enhanced or in some way facilitated by JavaScript's capability to deal with different objects.

JavaScript and Java Applets

Java, while JavaScript's namesake, has very little in common with JavaScript other than some structures in the way that it is coded. Java is a compiled language, and because it is not interpreted in the browser as is HTML and JavaScript, it must be loaded into an HTML page to be viewed in a browser. Like a graphic or SWF file, Java programs, called "applets" (little applications), can be placed anywhere in an HTML script. The following format is the general one to bring in a Java applet to an HTML page:

```
<applet code="appletName.class" name="anyName" width=w height=h>
```

The width and height variables set a block reserved for the Java applet. Measured in pixels, the block can take up just enough space for the applet, or it can take up a larger chunk that will be blank around the applet.

JavaScript to Check for Java-Enabled Browsers

I'll come back to the HTML surrounding a Java applet, but first you should have a way to find whether your browser's Java is enabled. You can save yourself a good deal of time and frustration, as well as that of the viewer who looks at your pages, by using JavaScript to see if Java is enabled. If Java is not enabled, no Java applet can check for that because, if it is disabled, the applet won't run. Therefore, JavaScript is essential to determine whether the user (or designer) has her Java enabled in the browser.

The general JavaScript technique for determining whether a browser has Java enabled uses this method with the `navigator` property to generate a Boolean outcome:

```
javaEnabled()
```

Hence, this line used in a conditional statement returns `true` or `false:`

```
navigator.javaEnabled()
```

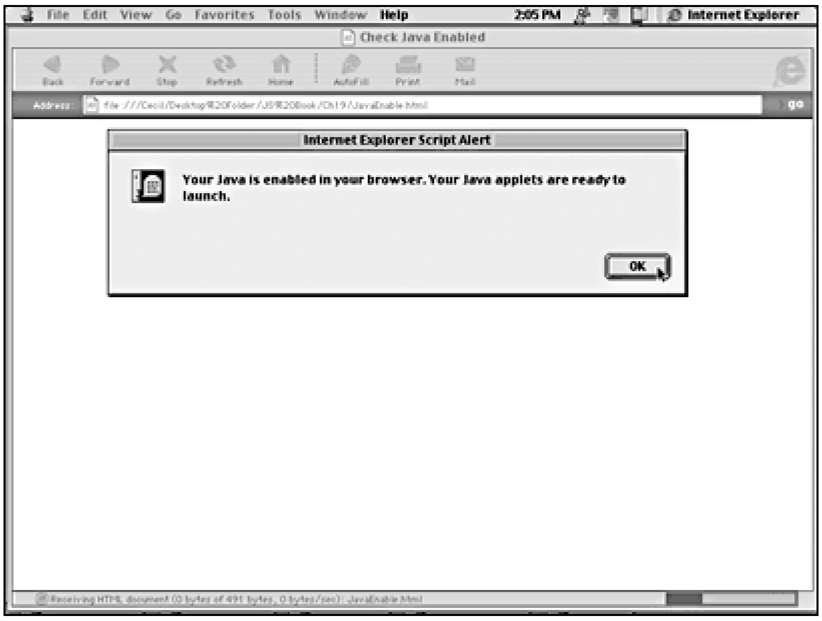
The following script demonstrates how the method can be used. (Try it out on your own browser with Java enabled and disabled.)

```
<html>
```

```
<head>
<title>Check Java Enabled</title>
<script language ="JavaScript">
var gottaJava="Your Java is enabled in your browser. Your Java
applets are ready to
launch.";
var lackaJava="Java is not enabled and until you enable Java in your
browser, none
of the Java applets will launch.";
function gotJava() {
     alert(gottaJava)
     }
function nadaJava() {
     alert(lackaJava)
     }
if(navigator.javaEnabled()) {
            gotJava();
     } else {
           nadaJava();
     }
</script>
</head>
<body>
The page is up.
</body>
</html>
```

Figure 19.1 shows what you can expect when your browser has Java enabled.

Figure 19.1. Using JavaScript to tell a user whether the browser has Java enabled can save a lot of frustration in looking at blank pages.



file:///Cecil/Desktop%20Folder/JS%20Book/Ch19/JavaEnable.html

Internet Explorer Script Alert

Your Java is enabled in your browser. Your Java applets are ready to launch.

Receiving HTML document (0 bytes of 491 bytes, 0 bytes/sec): JavaEnable.html

A Little Java

At this point, you will need a Java applet to see the role of JavaScript. You can find plenty of free Java applets on the web. For example, `java.sun.com` has lots of free applets. (Hint: Click on Applets, Freebie Applets). However, if you'd like to try a little Java yourself, follow this path:

Java.sun.com > Java Tutorial > Your First Cup of Java Win32|UNIX|Mac

Be sure to click on Win32, UNIX, or Mac instead of the general Your First Cup of Java link. Download the Java 2 Platform, Standard edition, and the Developer's Kit (all free!). When you have gone through the tutorial, try out the following Java applet. First, write the source code in a text editor (such as Notepad or SimpleText). Save the file as tooEZ.java. Note the different versions for Windows PC and Macintosh.

Windows:

tooEZ.java

```
import java.applet.*;
import java.awt.*;
public class tooEZ extends Applet {
      public void paint(Graphics g) {
           g.drawString("Java is a bit more demanding than
JavaScript.", 50, 25);
           g.drawString("You will find it used on servers as well.",
80, 50);
     }
}
```

Macintosh:

tooEZ.java

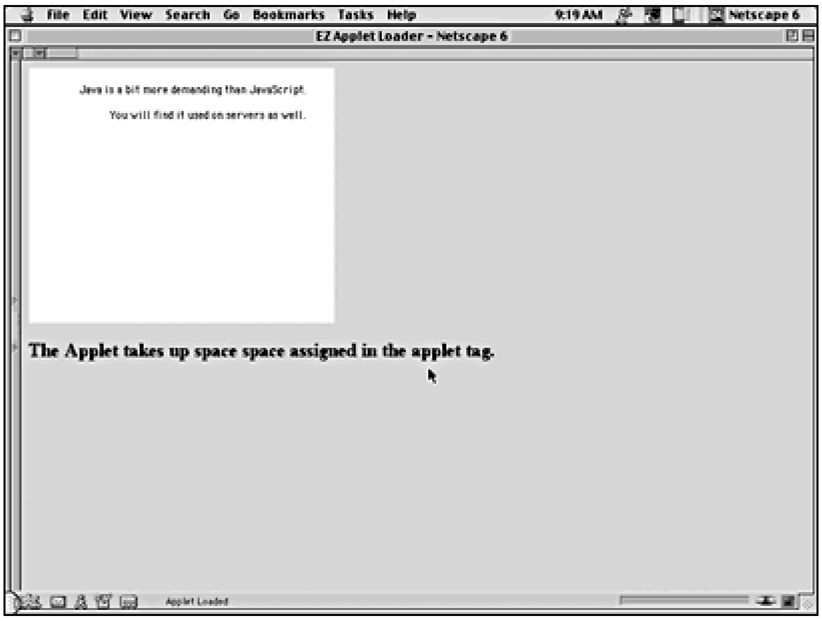
```
import java.applet.Applet;
import java.awt.Graphics;
public class tooEZ extends Applet {
      public void paint(Graphics g) {
            g.drawString("Java is a bit more demanding than
JavaScript.", 50, 25);
            g.drawString("You will find it used on servers as well.",
80, 50);
      }
}
```

Compile the tooEZ.java file to generate the tooEZ.class file. The class file is the file that you load into an HTML page. The following HTML page will load your applet and show it on the screen:

```
<html>
<head>
<title>EZ Applet Loader</title>
</head>
<body bgcolor="powderblue">
<applet name="javaDaba" code="tooEZ.class" width=300 height=250>
</applet>
<h3>The Applet takes up space assigned in the applet tag.</h3>
</body>
</html>
```

Note that the applet does not have the same background color as the rest of the HTML page. All font and background colors have to be generated in the Java applet. Figure 19.2 shows what you should see on your screen.

Figure 19.2. The block occupied by the Java applet moves the HTML text lower on the screen.


File Edit View Search Go Bookmarks Tasks Help
9:19 AM
Netscape 6
EZ Applet Loader – Netscape 6
Applet Loaded

Java is a bit more demanding than JavaScript.

You will find it used on servers as well.

The Applet takes up space space assigned in the applet tag.

JavaScript and Applets

When an applet is embedded in an HTML page, JavaScript can access it through an `applets[]` array in the `Document` object expressed as an element name or number. More importantly, JavaScript can address a Java applet's methods. Unfortunately, Netscape Navigator and Internet Explorer do not share a compatible way of dealing with Java or JavaScript in this context. Microsoft ActiveX is used to control applets in Internet Explorer, and LiveConnect is in charge in Netscape Navigator up to Version 4. Beginning with Version 6, Netscape changed the DOM so that the `applets[]` array does not access Java controls in the same way as *NN4.x* . So, the following discussion will focus on using *NN4.x* with JavaScript and Java objects in an HTML page. As tested, they ran on both *NN4.7* and *IE5.5* .

Calling Java Applet Methods from JavaScript

To see how JavaScript calls a Java applet's method, the next example makes a Java applet appear and disappear on the page. The two Java methods, `hide()` and `show()`, are standard methods of the Java `Applet` class. If you have applets that have their own methods, they can be called by JavaScript as well. These next two statements both call the applet named in the `<applet>` tag—once by its name and once by its element value:

```
document.javaDaba.hide()
```

and

```
document.applets[0].show()
```

However, the two methods, `hide()` and `show()`, are *not* part of the JavaScript DOM and so are demonstrably methods belonging to Java. Figures 19.3 and 19.4 show two buttons firing JavaScript functions to make the Java applet disappear and then reappear.

Figure 19.3. The left button hides the Java applet by launching a JavaScript function.

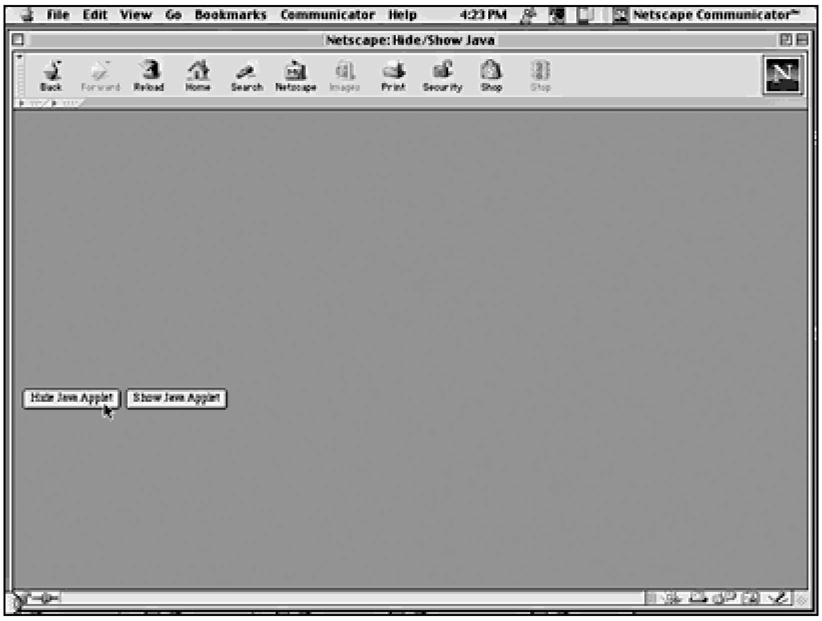
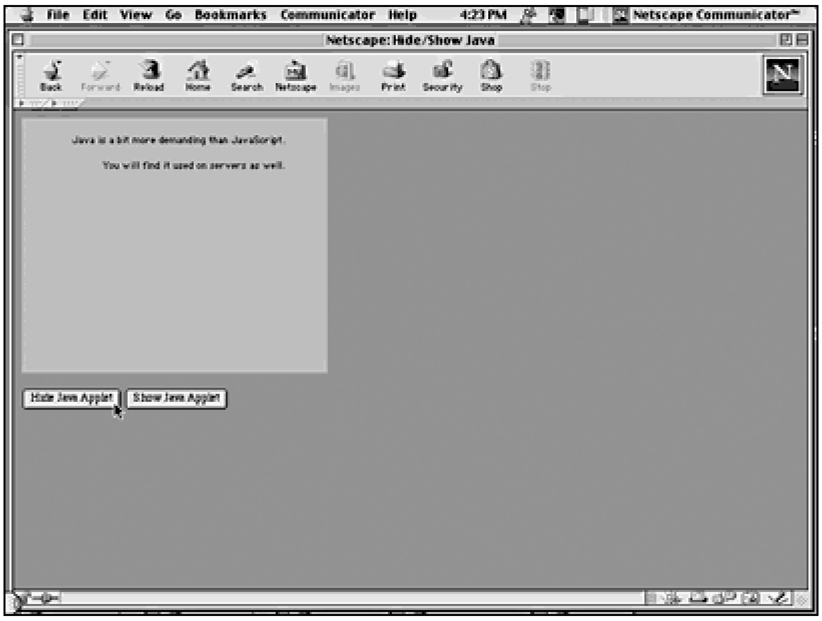


Figure 19.4. The right button makes the hidden Java applet reappear.



Java is a bit more demanding than JavaScript.

You will find it used on servers as well.

jsJava.html

```
<html>
<head>
```

```
<title>Hide/Show Java</title>
<script language ="JavaScript">
function hideJava( ) {
      document.javaDaba.hide( )
      }
function showJava( ) {
      document.applets[0].show( )
      }
</script>
</head>
<body bgcolor="cornflowerblue">
<applet name="javaDaba" code="tooEZ.class" width=300 height=250>
</applet>
<form name="stealth">
<input type=button value="Hide Java Applet" onClick="hideJava( )";>
<input type=button value="Show Java Applet" onClick="showJava( )";>
</form>
</body>
</html>
```

Many of the Java applets that were necessary for dynamic web pages have been replaced to some degree by newer applications of both JavaScript and animation programs that generate SWF files (such as Flash and LiveMotion). The fact that Java files are compiled and not interpreted by the browser did not diminish the fact that they had to be sent across the web, slowing down larger files. As a result, much of the Java programming has migrated to servers where Java "servelets" can effectively deal with server-side tasks.

JavaScript and ColdFusion

In Chapters 14– Chapter 19, different types of server-side or database applications were viewed working with JavaScript. JavaScript's main role was that of a preprocessor in a client-side environment. Another popular server-side program is one from Macromedia called ColdFusion. Like PHP, CGI, and ASP, ColdFusion works only with a special server that processes its code. However, unlike the other server-side languages discussed so far, ColdFusion has some JavaScript keywords that it adds to the JavaScript lexicon when used in a CF context. The keywords, however, are designed to work exclusively with CF and do not have a wider application.

If you decide to take up ColdFusion, you will be happy to learn that much of the *server-side* scripting is done in JavaScript. Unlike some of the other middleware examined in this book, JavaScript plays a server-side role in ColdFusion as well as a client-side role. The SQL commands will look virtually identical to MySQL and the SQL statements used in ASP pages.

Two JavaScript objects, `WddxSerializer` and `WddxRecordset`, are included in Cold-Fusion, each with several functions (methods). In the next sections, each of the objects is summarized along with each of the associated functions (methods).

WddxSerializer Object

As the name implies, this object contains functions to serialize JavaScript data structures. *Serializing* means that it generates a storable representation of a value. First, Web Distributed Data Exchange (WDDX) is a technology borrowed from XML. It facilitates complex data exchange between web programming languages (such as ColdFusion). So, a web site based in ColdFusion can share data with JavaScript. The data are translated into XML and then translated from XML into a language readable by ASP, PHP, or JavaScript. As you saw in Chapter 17, "Working with XML and JavaScript," JavaScript does not have a cross-platform XML interpreter. Internet Explorer uses ActiveX or `document.all` to bring data out of XML to the screen. What ColdFusion has done is add its own JavaScript object to make an equivalent transition. The only difference is that the WDDX technology facilitates the translation rather than ActiveX.

`WddxSerializer` supports four functions, but only the first, `serialize( )`, is likely to be used to any extent. The other can be used, but the data are more likely to be passed off using the `serialize( )` function:

- `serialize(rootOjb)` Creates a WDDX packet for the `WddxRecordset` object instance. On a server-side application, the following is a simple function to serialize the data:

```
•
•   function getTheData(data,formAlpha) {
•          setSerial= new WddxSerializer( ); //define object
•          dataGroup= setSerial.serialize(data); //establish
packet
•          if(dataGroup !=null) {
•                 formAlpha.value=dataGroup;
•          }else{
•                 alert("Data will not serialize")
•                 }
}
```

- `serializeVariable(name, obj)` Serializes a property of a structure. When the object is not a date, Boolean, string, number, or array, it is treated as a structure.
- `serializeValue(obj)` Serializes eligible data, which includes any JavaScript object, data, array, Boolean, number, string, or recordset.
- `write(string)` Is an internal function not typically called. It appends data to a serialized data stream.

WddxRecordset Object

The `WddxRecordset` object includes several functions (methods)for construction of WDDX recordsets. Unlike the `WddxSerializer`, the `WddxRecordset` creates recordsets rather than serializes existing ones. However, one of the methods, `wddxSerialize( )`, *does* serialize a recordset of the `WddxRecordset` object.

- `addColumn(name)` Adds a column to the recordset in a `WddxRecordset` instance. First the recordset is defined as an object in JavaScript, and then the `addColumn( )` method adds a named column. The following script segment shows using this and other methods associated with the `WddxRecordset` object:

```
•
•   customerSet= new WddxRecordset( );
•   customerSet.addColumn("salesDist");
•   customerSet.addRows(7);
customerSet.setField(2,"salesDist", distNum);
```

- `addRows(number)` Add the number of rows specified in the argument. Columns use names, and rows use numbers. Generally, the columns are fields and the rows represent records.
- `getField (row,col)` Returns the *element* (not value) in a specified position indicated by the row number and column name.
- `getRowCount( )` Works something like a `length` method, but returns the number of rows rather than the number of items in an object or array or characters in a string.
- `setField(row,col,value)` Specifies a value in a row number of a named column—for example:
-
  ```
  customerSet.setField(4, "salesDistrict",7);
  ```

  This line would set the fifth row (the first is 0) of the column (field) named salesDistrict with the value of 7. The value 7 could have been a numeric or string variable as well.

- `wddxSerialize` Serializes a recordset object.

While this book has emphasized client-side scripting with JavaScript, most of the same structures used on the client side can be used on the server side as well. As an added bonus, learning a new language such as ColdFusion is aided considerably by the fact that much of the script will be JavaScript.

JavaScript and ASP.NET

Microsoft's new .NET framework is a very different back end (if it indeed can be called that) than ASP. In fact, ASP.NET technology is not backward-compatible with ASP (see Chapter 15, "Using ASP with JavaScript"). The VB.NET scripting language has many similarities with VBScript, but that is due to the fact that VBScript has many similarities with Visual Basic, one of the core codes of ASP.NET. Included in ASP.NET codes, besides Visual Basic, are the following:

- XML
- CSS
- HTML
- DHTML
- C++
- C# (C-sharp)

At the time of this writing, ASP.NET was still in its infancy (beta), but it has generated a good deal of interest. However, at this stage, JavaScript does not seem applicable. The C# language was designed to be a friendlier version of Java, but it is compiled and so is not a substitute for JavaScript. A combination of C# and Visual Basic, though, might be seen as the best way of handling objects previously done by JavaScript. Working in an environment of web forms for developing code, it is difficult to discern what role, if any, is to be played by JavaScript. On one hand, JavaScript could play the same role in ASP.NET that it does with ASP: that of an input preprocessor. On the other hand, preprocessing may be done in the ASP.NET web form itself.

ASP.NET is not backward-compatible with ASP (or ASP pages written in VBScript), but this does not mean that it is or is not compatible with JavaScript. Because Microsoft has included DHTML as a code that works with ASP.NET, you might be led to believe that, within the DHTML, JavaScript would be welcomed, if as nothing more than a preprocessor.

Summary

JavaScript is a Jack-of-all-trades scripting language. It works well with HTML, DHTML, and most middleware languages such as PHP, CGI, and ASP. What's more, as seen in this chapter, JavaScript works with compiled languages such as Java to act as an event handler for methods and as a detector of a browser's capability to detect Java.

ColdFusion has integrated JavaScript into the heart of its language. Not only can ColdFusion server-side scripts include JavaScript, but ColdFusion has added keywords to the JavaScript lexicon for enhancing its role as a database middleman.

Microsoft's ASP.NET, on the other hand, might not have a role for JavaScript. At this stage of ASP.NET's development, it is difficult to say with certainty, but most of JavaScript's traditional functions have been solved with other coding options. Nevertheless, when JavaScript was first introduced, no one had any idea how key it would be come to the Internet, and ASP.NET could yet find it necessary and useful to work in a JavaScript environment.

Appendix Example Glossary

THIS GLOSSARY IS FOR A QUICK LOOK-UP of different keywords, statements, and terms used in standard ECMA-262 JavaScript, with clear examples provided. You will not find words that either Microsoft or Netscape developed outside the ECMA-262 standard, no matter how helpful these terms are. Also, you will not find a standard "Reference" section here. If that's what you want, you can find the mother of all JavaScript references at http://developer.netscape.com/evangelism/docs/reference/ecma/Ecma-262.pdf.

The European Computer Manufacturer's Association (ECMA) is *the* definitive authority on JavaScript, and its 188-page reference section includes all of the technical material about JavaScript. (See also http://www.ecma.ch.)

This glossary's function is to make it easy to find a term, see what it does, and see an example of how to write it. It's quick and simple, and it will help you get a statement correctly written. To keep this glossary clear, the examples are short and to the point, and the definitions are nontechnical but functional. Some examples are shorter than others, depending on what you need to use them effectively. Objects and their methods are listed by the nature of the object and are followed by a property or method. For example, the `string` object (which is almost never called "string") will include `string.length`. Typically, the length of a string will be found in an iterative loop such as this:

```
for(counter=0;counter < cusName.length; counter++) {....
```

The string's name is `cusName`, which is a far more likely string name than `string`. However, to make it simple to find what properties and methods are associated with strings, I use `string.xxxx` to make it simple to find things alphabetically in the glossary. Moreover, some properties and methods are part of an object *of an* *object.* For example, `form` has several different associated elements, such as `Input`, so you will see "`formInput.text`" and "`formInput.button`" entries because they are all types of `<input>` options. Also, several objects have similar or identical properties. For example, you will find "`history.length`" and "`array.length`" entries, among others. So, to help keep things clear, it makes more sense to alphabetize the objects using clear names rather than the properties or methods that might occur in several different objects.

`alert(value)`

Opens an alert box on the screen in the browser.

Example:

```
var fname="Linda"
alert("Hello " + fname)
```

`anchor`

Creates an anchor in a document.

Example:

```
var fish = "trout"
document.write(fish.anchor("fresh_water"))
```

`applet`

An applet in the current HTML page. May be expressed as a name or an array.

Example:

```
document.applets[3].hide( )
```

`Array( )`

A multielement array object. The keyword `Array( )` is a constructor object. Optionally, you can enter the number of array elements.

Example:

```
var group=new Array(5);
var gang = new Array("Fuzzy", "Willie", "Sleepy", "Homer")
```

`array.concat( )`

Adds new elements that do not become a permanent part of the array.

Example:

```
var dogs = new Array("Rottie", "Sheltie")
dogs.concat("Swissy", "Beagle")
alert(dogs.concat("Swissy", "Beagle"))
//Output = Rottie, Sheltie, Swissy, Beagle
```

`array.join( )`

Elements in an array are made into a single concatenated string with an optional separator.

Example:

```
var dogs = new Array("Rottie", "Sheltie")
alert(dogs.join("—"))
//Output = Rottie—Sheltie
```

`array.length`

Returns the length of an array.

Example:

```
var cities = new Array ("Paris", "London", "Los Angeles",
"Bloomfield");
var nCities = cities.length;
//nCities = 4
```

`array.reverse( )`

Reverses the order of elements in an array.

Example:

```
var geekLetters = new Array("alpha", "beta", "gamma");
alert(geekLetters.reverse( ));
//Output = gamma, beta, alpha
```

`array.slice(s,e)`

Returns a segment of an array beginning with *s* and ending with e.

Example:

```
var cities = new Array ("Paris", "London", "Los Angeles",
"Bloomfield");
var nCities = cities.splice(1,2);
alert(nCities)
//Output= London, Los Angeles
```

`array.sort( )`

Puts array elements into alphabetical order.

Example:

```
var cities = new Array ("Paris", "London", "Los Angeles",
"Bloomfield");
var nCities = cities.sort( );
//nCities = Bloomfield,London, Los Angeles, Paris
```

`array.toString( )`

Converts all array elements to a single string.

Example:

```
var deli = new Array ("Bagels", "Lox", "Dill Pickles", "Cream
Cheese");
var food = deli.toString( );
// food = single string — Bagels,Lox,Dill Pickles,Cream Cheese
```

`button`

See [**formInput.button.value**]

`checkbox`

See [**formInput.checkbox.value**]

`clearTimeout( )`

See [**window.clearTimeout( )**]

`close( )`

See [**window.close( )**]

`closed`

See [**window.closed**]

`confirm( )`

See [**window.confirm(q)**]

`Date( )`

Date and time object.

Example:

```
var today=new Date( );
document.write(today);
//Screen shows: Sat Jul 28 10:52:44 GMT-0400 (2001)
```

`Date.getDate( )`

Day of the month, as a number between 1 and 31.

Example:

```
var today=new Date( );
alert(today.getDate( ));
//Value between 1 and 31
```

`Date.getDay( )`

Day of the week, expressed as a value 0 to 6, with Sunday being 0.

Example:

```
var today=new Date( );
alert(today.getDay( ));
//Value between 0 and 6
```

`Date.getFullYear( )`

Returns the current year.

Example:

```
var today=new Date( );
alert(today.getFullYear( ));
//Actual year such as 2003
```

`Date.getHours( )`

Returns 24 hours, from 0 to 23 (midnight is 0).

Example:

```
var today=new Date( );
```

```
alert(today.getHours( ));
//2 PM returns 14
```

`Date.getMilliseconds( )`

Returns 0 to 999 in milliseconds.

Example:

```
var today=new Date( );
var now = today.getMilliseconds( )
document.write(now);
//Enter script and press Re-load on browser to see changes
```

`Date.getMinutes( )`

Gets the current minute of the hour, between 0 and 59.

Example:

```
var today=new Date( );
var now = today.getMinutes( )
document.write(now);
```

`Date.getMonth( )`

Gets the current month, from 0 to 11 (January is 0).

Example:

```
var today=new Date( );
var now = today.getMonth( )
document.write(now);
```

`Date.getSeconds( )`

Returns the current seconds, 0 to 59.

Example:

```
var today=new Date( );
var now = today.getSeconds( )
document.write(now);
```

`Date.getTime( )`

The time between the Date( ) value and January 1, 1970.

Example:

```
var today=new Date();
var now = today.getTime( )
document.write(now);
//Some value like 996335075147 appears
```

`Date.getTimezoneOffset( )`

The difference, in minutes, in local time and UTC (universal time, or Greenwich Mean Time).

Example:

```
var today=new Date( );
var now = today.getTimezoneOffset( )
document.write(now/60);
//Shows difference in hours because to division by 60.
```

`Date.getUTCDate( )`

Returns the UTC day of month (1–31).

Example:

```
var today=new Date( );
var now = today.getUTCDate( )
```

`Date.getUTCDay( )`

Returns the UTC day of week (0–6).

Example:

```
var today=new Date( );
var now = today.getUTCDay( )
```

`Date.getUTCFullYear( )`

Returns the UTC year (such as 2003).

Example:

```
var today=new Date( );
var now = today.getUTCFullYear( )
```

`Date.getUTCHours( )`

Returns UTC hours (0–23).

Example:

```
var today=new Date( );
var now = today.getUTCHours( );
```

`Date.getUTCMilliseconds( )`

Returns milliseconds of a UTC date.

Example:

```
var today=new Date( );
var now = today.getUTCMilliseconds( )
```

`Date.getUTCMinutes( )`

Returns UTC minutes (0–59).

Example:

```
var today=new Date( );
var now = today.getUTCHours( );
```

`Date.getUTCMonth( )`

Returns the UTC month (0–11).

Example:

```
var today=new Date( );
var now = today.getUTCMonth( );
```

`Date.getUTCSeconds( )`

Returns UTC seconds (0–59).

Example:

```
var today=new Date( );
var now = today.getUTCSeconds( );
```

`Date.getYear( )`

This function contains bugs. Use `getFullYear( )` instead.

`date.setDate( )`

Sets the day of the month between 01 and 31. (It does not change your computer's internal clock.)

Example:

```
var today=new Date( );
today.setDate(31);
var myTime=today.getDate( );
//myTime = 31 no matter what the day is
```

`date.setFullYear( )`

Sets the year to a specified year.

Example:

```
var today=new Date( );
var now=today.getFullYear( );
today.setFullYear(1992);
alert("That was " + (now - today.getFullYear()) + " years
ago." );
```

`date.setHours( )`

Sets the hours of the day from 0 to 23.

Example:

```
var today=new Date( );
today.setHours(15);
document.write(today.getHours( ));
```

`date.setMilliseconds(m)`

Sets m to an integer between 0 and 999. (Values greater than 999 reduce to the three rightmost values; 5432 reverts to 432.)

Example:

```
var today=new Date( );
today.setMilliseconds(999);
document.write(today.getMilliseconds( ));
```

`date.setMinutes( )`

Sets minutes between 0 and 59.

Example:

```
var today=new Date( );
today.setMinutes(43);
document.write(today.getMinutes( ));
```

`date.setMonth( )`

Sets the month, between 0 and 11.

Example:

```
var today=new Date( );
today.setMonth(5); //Sets June
document.write(today.getMonth( ));
```

`date.setSeconds( )`

Sets seconds, between 0 and 59.

Example:

```
var today=new Date( );
today.setSeconds(54);
document.write(today.getSeconds( ));
```

`date.setTime( )`

Sets the time in milliseconds relative to January 1, 1970.

Example:

```
var today=new Date( );
today.setDate(996338910155);
//Sets July 28, 2001
```

`date.setUTCDate( )`

Sets the UTC day of the month (1-31).

Example:

```
var today=new Date( );
today.setUTCDate(22);
document.write(today.getUTCDate( ));
```

`date.setUTCFullYear( )`

Sets the UTC full year (such as 2020).

Example:

```
var today=new Date( );
today.setUTCFullYear(2002)
```

`date.setUTCHours( )`

Sets UTC hours (0–23).

Example:

```
var today=new Date( );
today.setUTCHours(17)
```

`date.setUTCMilliseconds( )`

Sets UTC milliseconds (0–999).

Example:

```
var today=new Date( );
today.setUTCMilliseconds(999);
document.write(today.getUTCMilliseconds( ));
```

`date.setUTCMinutes( )`

Sets minutes from 0 to 59.

Example:

```
var today=new Date( );
today.setUTCMinutes(45);
```

`date.setUTCMonth( )`

Sets UTC months from 0 to 11.

Example:

```
var today=new Date( );
today.setUTCMonth(5);
```

`date.setUTCSeconds( )`

Sets UTC seconds from 0 to 59.

Example:

```
var today=new Date( );
today.setUTCSeconds(5);
```

`date.toLocaleDateString( )`

Conversion of date to string.

Example:

```
var today=new Date( );
var ls=today.toLocaleString( );
document.write(ls);
//formatted as Jul 28 14:59:29 2001
```

`date.toString( )`

Conversion of date to string. Note differences in format between toString( ) and toLocaleString( ).

Example:

```
var today=new Date( );
var ls=today.toString( );
document.write(ls);
//formatted as Sat Jul 28 15:03:55 GMT-0400 (2001)
```

`date.toUTCString( )`

Conversion of data to a UTC string.

Example:

```
var today=new Date( );
var ls=today.toUTCString( );
document.write(ls);
//formatted as Sat, 28 Jul 2001 19:06:12 GMT
```

`date.valueOf( )`

Date converted to a number.

Example:

```
var today=new Date( );
var ls=today.valueOf( );
document.write(ls);
```

`document`

A web (HTML) page. References to the document object are to its properties and methods. It may also be referenced as part of a window.

Example:

```
window.document
```

`document.alinkColor`

Color when link is selected.

Example:

```
document.alinkColor="green";
```

`document.anchors[ ]`

Anchors in the current document as the named object or element.

Example:

```
var alpha=document.anchor[4]
```

`document.applets[ ]`

Reference to applets in the current document as a named object or element.

Example:

```
var alpha=document.applets[0]
```

`document.bgColor`

Background color of a page.

Example:

```
document.bgColor="'#ff00ff"
```

`document.close( )`

Closes an open document when writing HTML code using JavaScript.

Example:

```
document.close( )
```

`document.cookie`

The cookie on an HTML page.

Example:

```
var crumbs = document.cookie;
//Read value of cookie into variable "crumbs"
```

`document.domain`

Specifies the domain from which one window can read another window.

Example:

```
document.domain="sandlight.com";
```

`document.embeds[ ]`

Array of objects of data embedded in an HTML page.

Example:

```
var embedBugs = document.embeds.length
```

`document.fgColor`

A document’s default text color.

Example:

```
document.fgColor="cornflowerblue";
```

`document.forms[ ]`

The form object of an HTML page. All forms constitute elements of a form’s array object.

Example:

```
document.forms[0].reset( );
```

`document.images[ ]`

The image object of an HTML page. All images on the page are part of the image object array.

Example:

```
var sizeImages=document.images.length;
```

`document.lastModified`

Date of last modification.

Example:

```
var fixUp = document.lastModified;
document.write("Last modified by Al => " + fixUp);
```

`document.linkColor`

Sets unvisited link color.

Example:

```
document.linkColor="peru"
```

`document.links`

An HTML page’s link array object. (Properties of a page’s links can be displayed after a page is fully loaded.)

Example:

```
function showMe( ) {
var alpha=document.links.length;
alert(alpha);
}
```

`document.location`

Access URL.

Example:

```
document.location="http://www.sandlight.com";
//link to www.sandlight.com
```

See also [**document.URL**]

`document.open( )`

Opens a new document. Typically used for writing HTML pages from script in JavaScript.

Example:

```
document.open( );
```

`document.plugins[ ]`

See [**document.embeds[ ]**]

`document.referrer`

The page that linked *to* the current page. Requires a link *from* a previous page.

Example:

```
var homie = document.referer;
document.location=homie;
```

`document.title`

Current HTML page's title.

Example:

```
alert(document.title);
```

`document.URL`

URL of specified page. (Replaces `document.location`.)

Example:

```
Document.URL="http://www.sandlight.com"
```

`document.vlinkColor`

Specifies the color of visited links.

Example:

```
document.vlinkColor="#ff00ff";
```

`document.write( )`

Used for both sending text to a page and writing an HTML document.

Example:

```
document.write("<b>'This is bold' </b>");
```

`document.writeln( )`

Same as `document.write`, with an added carriage return.

Example:

```
document.writeln("This is the first line.")
document.writeln("This is on another line.")
```

`escape( )`

Codes string for sending in a form or email.

Example:

```
var alpha="Happy Birthday, Pat";
var sendMe=escape(alpha);
document.write(sendMe);
//Returns = Happy%20Birthday%2C%20Pat
```

`eval( )`

Evaluates an expression and puts it into a string.

Example:

```
var alpha=eval(Math.sqrt(16));
```

```
alert(alpha);
//Output = 4
```

`form`

Input form treated as an array object in JavaScript.

Example:

```
document.forms[2].reset( )
```

`form.elements[ ]`

All of the input and textarea elements in a form container. Each is treated as an element of form array object and is addressed by element name or number.

Example:

```
var output = document.forms[0].elements[2].value;
var output = document.customers.lastNames.value;
```

`form.length`

The numbers of elements in a form container.

Example:

```
var howLong = document.forms[3].length;//Number of elements in
form.
var myForm = document.forms.length; //Number of forms in
document
```

`form.reset( )`

Clears all data from a form.

Example:

```
Function clearEm( ) {
document.forms[0].reset( );
}
```

`form.reset.value`

Current value of a Reset button. The value is placed in an `<input>` tag.

Example:

```
var alpha=document.forms[0].wipe.value; //wipe is name of reset
button
```

form.textarea.value

Returns the contents of a textarea.

Example:

```
var Texas = document.forms[0].fred.value;
//The text area's name is "fred."
//The variable Texas contains the contents of the textarea
//named "fred."
```

formInput.button.value

Returns or changes value assigned to a button.

Example:

```
var button = document.forms[0].butNow.value;
//Returns the name on the button!
```

formInput.checkbox.checked

Generates a Boolean value on whether a check box is checked.

Example:

```
//Checkbox object named "chuck"
if (document.forms[0].chuck.checked) {....
```

formInput.checkbox.defaultChecked

Boolean read-only to determine whether a check box is initially checked.

Example:

```
var alpha = document.forms[0].chuck.defaultChecked;
if(alpha) {
alert("It's checked from the beginning.") }
```

formInput.checkbox.value

Returns `on` unless a specific value is assigned in the `Value` attribute in the `<input>` container.

Example:

```
var alpha = document.forms[0].chuck.value
```

`formInput.name`

Returns the name of a form element.

Example:

```
var alpha=document.forms[0].elements[4].name;
//Places the name of the fifth element into alpha
```

`formInput.password.value`

The value of a password, as defined in the `value` attribute of the form element.

Example:

```
var openUp = document.forms[0].elements[4].value;
if(openUp=="reallySecret") {....
```

`formInput.radio.checked`

A Boolean value to determine whether a radio button is checked.

Example:

```
var alpha=document.forms[0].elements[2].checked;
if(alpha) {
alert(":It’s checked");
} //elements[2] is one of three radio buttons
```

`formInput.radio.value`

The value of the value attribute is assigned in the `<input>` tag in the HTML page. If no value is assigned, the value returned is `on`, whether the button is checked or not. While all the names of radio buttons in a form should be the same, the values are typically different.

Example:

```
var alpha=document.forms[0].elements[3].value;
```

```
//elements[3] is a radio button input element
```

`formInput.submit.value`

The value of the Submit button assigned in the `<input>` tag, or `Submit Query` if no value is assigned.

Example:

```
var alpha=document.forms[0].jack.value;
//"jack" is the name of the submit button.
```

`formInput.text.value`

The current value of a text field or a value to be assigned.

Example:

```
var alpha=document.forms[0].announce.value;
alpha = "Read this."; //String assigned to text field
"announce".
```

`formInput.type`

Returns `select-one` or `select multiple`, depending on the contents of the `<select>` tag.

Example:

```
var alpha=document.forms[0].nancy.type;
alert(alpha);
//name is the select element's name.
```

`formInput.value`

The value of a form input element.

Example:

```
var fullName=document.forms[0].name.value;
```

`forms[n].reset( )`

Clears all text forms.

Example:

```
<body onLoad=document.forms[0].reset( )
```

formSelect.length

The length refers to the number of options in a select property.

Example:

```
var numOps=document.forms[0].chooser.length;
alert("You have " + numOps + " options.");
//"chooser" is the name of the select form element.
```

formSelect.options

Options properties in a select object.

Example:

```
var selections=document.forms[0].chooser;
var selOp=selections.options.length;
```

formSelect.selectedIndex

Element value of the selected option.

Example:

```
var selections=document.forms[0].chooser; //chooser is select
obj.
var selOp=selections.selectedIndex;
alert(selOp)//Element value of selected options appears
```

formSelect.type

Either `select one` or `select-multiple` types.

Example:

```
var selections=document.forms[0].chooser;
var selType=selections.type
alert(selType) //chooser is select object
```

frame

See [window.frames[ ]]

`function`

A delayed operation fired by an event handler.

Example:

```
function showView(msg) {
alert(msg); //Statements go here.
} //Function terminated by closing curly brace
```

`function.toString( )`

Places the ouput of a function into a string format.

Example:

```
function showMe(msg) {
alert(msg)
}
var buzz=showMe("I am a function").toString;
alert(buzz);
```

`history.back( )`

Goes to the previously visited URL that was viewed *before* the current page in the current session.

Example:

```
window.history.back( );
```

`history.forward( )`

Goes to the previously visited URL that was viewed *after* the current page in the current session.

Example:

```
window.history.forward( )
```

`history.go( )`

Goes forward or backward a specified number of pages. Positive numbers go forward, and negative numbers go backward.

Example:

```
window.history.go (2) // Two pages forward
window.history.go (-3) // Three pages back
```

image

Reference to an image in an HTML page. All embedded images are considered part of an image array.

Example:

```
var tUp=new Image()
tUp.src="targetUp.jpg"
function showTarget() {
document.images[1].src=tUp.src
}
```

image.border

Returns the width of the border specified in an `<img>` tag.

Example:

```
var bdr=document.images[0].border;
alert(bdr); //bdr is the width of the border
```

image.complete

Boolean value of whether the browser has completed loading an image.

Example:

```
var upYet=document.images[1].complete;
if(upYet) {
alert("Image is loaded");
}
```

image.height

Returns the value of the height HTML attribute.

Example:

```
var grH = document.flame.height
//flame = graphic name attribute in <img> tag.
```

image.hspace

Returns the number of blank pixels on the left and right of the image.

Example:

```
var xPixes = document.flame.hspace
//flame = graphic name attribute in <img> tag.
```

`image.name`

Name attribute of an `<img>` tag.

Example:

```
//Name in <img> tag is "flower"
document.flower
```

`image.src`

The filename (or URL) of the image to appear in the browser. This can be changed to allow different images to occupy the same position. The first URL is defined in the HTML `<img>` tag in the SRC attribute.

Example:

```
var rollin= new Image( );
rollin.src="pix/nextOne.jpg";
document.images[3].src=rollin.src;
```

`image.vsapce`

Returns the number of blank pixels above and below an image.

Example:

```
var vPixes = document.flame.vspace
//flame = graphic name attribute in <img> tag.
```

`image.width`

Returns the width of the image set in the `<img>` tag's `Width` attribute.

Example:

```
var w=document.images[0].width;
```

`isFinite(n)`

Returns a Boolean value of whether the number is finite.

Example:

```
var littleNumber = 123
if(isFinite(littleNumber)) {
alert("We can afford it!")
}
```

`isNaN(v)`

Returns a Boolean value of whether the value is a number.

Example:

```
if(isNaN(price) {
alert("Your entry is not a number)
}
```

`link`

A link in an HTML page. Each link is an element in the links array belonging to the document object.

Example:

```
function numLinks( ) {
var nl = document.links.length;
alert("You have " + nl " links in your page.");
}
```

`location`

The browser location and a control of that location. Each of the following properties of location make up a part of a complete URL. The following semifictitious URL contains all of the properties in location: http://www.sandlight.com:1944/js/seeker.html?script#side.

- `location.hash #side`
- `location.host www.sandlight.com:1944`
- `location.hostname www.sandlight.com`
- `location.href The entire URL`
- `location.pathname js/seeker.html?script#side`
- `location.port 1944`
- `location.protocol http://`

- `location.search ?script`

All of the location properties are read/write and can be used to access any of the properties of the `location` object.

See also [**document.location**]

```
location.reload( )
```

Reloads the current page.

Example:

```
Location.reload( )
```

```
location.replace()
```

Replaces the current page with a new page, but without keeping a history.

Example:

```
location.replace(http://www.sandlight.com)
```

Math Constants

All math constants have a single value, as noted here. You can place them into variables or use the constants themselves wherever needed. Use the following example format:

```
var cirArea = Math.PI * (radius * radius)
```

Table G.1 shows the constants, their meaning, and their value.

Table G.1. JavaScript Math Constants

Name	Meaning	Value
`Math.E`	Constant e	`2.718281828459045`
`Math.LN10`	Constant $\log^{e}$ 10	`2.302585092994046`
`Math.LN2`	Constant $\log^{e}$ 2	`.6931471805599453`
`Math.LOG10E`	Constant $\log^{10}$ e	`.4342944819032518`
`Math.LOG2E`	Constant $\log^{2}$ e	`1.4426950408889634`
`Math.PI`	Constant pi	`3.141592653589793`
`Math.SQRT1_2`	Constant 1 divided by the square root of 2	`.7071067811865476`
`Math.SQRT2`	Constant square root of 2	`1.4142135623730951`

Math Functions

All built-in JavaScript math functions expect some type of argument, some with a range. However, their format is the same as for math constants and can be placed into variables or objects, as can other functions. Table G.2 provides the basics on math functions.

Table G.2. Math Functions and Parameters

Name	Meaning	Arguments
`Absolute value`	Math.abs( )	Any positive or negative number
`Math.acos( )`	Arc cosine	–1.0 to 1.0
`Math.asin( )`	Arc sine	–1.0 to 1.0
`Math.atan( )`	Arc tangent	Any positive or negative number
`Math.ceil( )`	Round up number	Any positive or negative number
`Math.cos( )`	Cosine	Any angle measured in radians*
`Math.exp( )`	e^x computed	Any expression or value to be used as exponent
`Math.floor( )`	Round down number	Any positive or negative number
`Math.log( )`	Natural logarithm	Any positive value
`Math.max(n1,n2)`	Larger of two values	Any two values
`Math.min(n1,n2)`	Smaller of two values	Any two values
`Math.pow(n1,n2)`	Computes the power of	First value to the power of the second value
`Math.random( )`	Returns a random number	Returns a number between 0.0 and 1.0
`Math.round( )`	Rounds to the nearest whole	Any value
`Math.sin( )`	Sine	Any angle measured in radians*
`Math.sqrt( )`	Square root	Any positive number
`Math.tan( )`	Tangent	Any angle measured in radians*

To convert an angle to a radian, use this method:

```
var radian = angle * (Math.PI * 2) / 360
```

`navigator`

Browser object (not just Netscape Navigator).

`navigator.appCodeName`

Returns Mozilla for both IE and NN.

Example:

```
var ncode=navigator.appCodeName
alert(ncode) //See Mozilla!
```

`navigator.appName`

Returns browser name (Microsoft Internet Explorer or Netscape).

Example:

```
var nName=navigator.appName
alert(nName)
```

navigator.appVersion

Returns the version of the browser. However, both Netscape and Microsoft have version numbers that do not match the version on their browsers. Version 6 of IE returns 4.0, and Version 6.1 of NN returns 5.0. Also returns the encryption and platform.

Example:

```
var nVer=navigator.appVersion
alert(nName)
```

navigator.javaEnabled( )

Checks to see whether Java is enabled in your current browser.

Example:

```
if(!navigator.javaEnabled( )) {
alert("Enable your browser for Java!")
}
```

navigator.platform

Returns the platform version (such as Wind32 or MacPPC).

Example:

```
var nPlat=navigator.platform
if(nPlat==Win32) {
document.fgcolor="cornflowerblue";
}
```

navigator.userAgent

Returns a combination of the code name and version (such as *Mozilla/5.0* [Macintosh; U; PPC; en-US; rv:0.9.1] Gecko/20010607 *Netscape6/6* .1b1).

Example:

```
var uAgnt = navigator.userAgent;
alert(uAgnt);
```

Number Constants

Like the math constants, the number constants have fixed values, even though many of the values must be represented by limit values. Table G.3 shows JavaScript's number constants.

Table G.3. Number Constants

Name	Meaning	Value
`Number.MAX_VALUE`	Largest number possible	`1.7976931348623157e+308`
`Number.MIN_VALUE`	Smallest number close to 0	`5e-324`
`Number.NaN`	Not-a-number value	Any non-numeric value
`Number.NEGATIVE_INFINITY`	A negative value greater than the highest value that JavaScript can represent	Negative maximum value plus minimum value
`Number.POSITIVE_INFINITY`	A positive value greater than the highest value that JavaScript can represent	Positive maximum value plus minimum value

Number Methods

A single `Number` method is available in JavaScript.

`numberObj.toString( )`

Converts a number object to a string.

Example:

```
var valWord=Number.MAX_VALUE;
valWord.toString();
```

`object`

A compound data type in which all other objects inherit the behavior of the object.

Example:

```
var kennel = new Object( );
kennel.Bigdogs = "Large breed dogs."
Kennel.Bigdogs.wolf = "Irish Wolfhounds"
```

`object.constructor`

A read-only reference to a type of object function used as constructor. For example, if an object is used as a constructor, `function Object()` is returned; if an array is used, function `Array()` is returned.

Example:

```
var kennel = new Object( );
document.write(kennel.constructor)
//return — function Object( ) { [native code] }
```

`object.toString( )`

Generally an automatic conversion in JavaScript, the method can clarify conversions.

Example:

```
var hotel = Excelsior.toString( ) //Excelsior is an existing
object
alert(hotel)
```

`object.valueOf( )`

Returns the object or its primitive value, but usually the object. Typically returns only the object itself. (Rarely used.)

Example:

```
var showOut = Excelsior.valueOf( );
document.write(showOut) // output=function valueOf( ) { [native
code] }
```

`parseFloat( )`

Converts a string to a floating-point number.

Example:

```
var strNum = "123.45";
var realNum = parseFloat(strNum); //realNum is floating point.
```

`parseInt( )`

Converts a string to an integer (rounded down).

Example:

```
var strNum = "123.45";
var intNum = intFloat(strNum);
```

`screen.availHeight`

Returns the vertical screen size in pixels.

Example:

```
var upScreen = screen.availHeight
```

`screen.availWidth`

Returns the available horizontal screen size in pixels.

Example:

```
var acrossScreen = screen.availWidth
```

`screen.colorDepth`

Returns the number of bits per pixel. Most modern computers provide 32-bit color.

Example:

```
var colorPix = screen.colorDepth;
```

`screen.height`

Returns the actual height of the screen. (It is different from `availHeight` in that it includes all space occupied by icon bars.)

Example:

```
var allScreenHi = screen.height;
```

`screen.width`

Returns the actual width of the screen.

Example:

```
var allScreenWide = screen.width;
```

string constructor

String objects are created using the `String( )` constructor.

Example:

```
var alpha=new String("Testing");
```

string.big( )

String is output in `<big>` format.

Example:

```
var alpha=new String("Testing");
document.write(alpha.big( ));
```

string.blink( )

String is output in `<blink>` format.

Example:

```
var alpha=new String("Testing");
document.write(alpha.blink( ));"
```

string.bold( )

String is output in `<b>` format.

Example:

```
var alpha=new String("Testing");
document.write(alpha.bold( ));
```

string.charAt(p)

Returns a character in a string at position `p`.

Example:

```
var alpha=new String("willie@harlemHome.org");
      for(var counter = 0; counter < alpha.length; counter ++)
{
```

```
                  if(alpha.charAt(counter) == "@") {
                  var flag=1;
            } //Sets a flag variable if the @ is found
      }
```

`string.charCodeAt(p)`

Returns the ASCII code of character at position `p`.

Example:

```
var alpha = new String("Fancy Characters &%$#@")
var asKey = alpha.charCodeAt(19);
```

`string.concat(s1,s2,sx)`

Concatenates strings in an argument to a string object.

Example:

```
var goof = new String("Mo");
var goofs = goof.concat("Larry","Curly","Shep");
document.write(goofs);
```

`string.fixed( )`

Sets font to <TT> style.

Example:

```
var alpha=new String("Testing");
document.write(alpha.fixed( ));
```

`string.fontcolor( )`

Assigns a font color to the string.

Example:

```
var alpha=new String("Color Me!");
document.write(alpha.fontcolor("pink"));
```

`string.fontsize( )`

Assigns a font size using HTML’s sizing units (1–7).

Example:

```
var alpha=new String("I'm Big");
document.write(alpha.fontsize(7));
```

`string.fromCharCode(c1,c2,cx)`

Creates a string from ASCII or Unicode character values.

Example:

```
var valentine=String.fromCharCode(76,79,86,69); //Note lack of
'new'
document.write(valentine);
```

`string.indexOf(s,st)`

Locates the first occurrence of substring (`s`) in a string, with an optional start (`st`) position. The initial position is `0`.

Example:

```
var alpha="Lots of characters."
var sIO = alpha.indexOf("char");
document.write("The substring begins at position " + sIO +
".");););
```

`string.italics( )`

Creates an italicized font.

Example:

```
var alpha=new String("I\'m from Rome!");
document.write(alpha.italics( ));
```

`string.lastIndexOf(s,st)`

Searches for the first occurrence of the substring beginning with the last character. The initial position is `0`.

Example:

```
var alpha="To be or not to be."
var lindx = alpha.lastIndexOf("be");
document.write("The last instance of the substring begins at
position " + lindx + ".");
```

`string.length`

Returns the length of the string.

Example:

```
var myName =new String("Rooty Judy Hooty");
var nameNum =myName.length -2
document.write("Her name is " + nameNum + " characters long.")
//Subtracted 2 for spaces;
```

`string.link(url)`

Creates a link to the specified URL.

Example:

```
var hookUp =new String("Treasure Island");
document.write(hookUp.link("http://www.sandlight.com"))
```

`string.match(re)`

Matches a string with one or more regular expressions. (Use Perl regular expression format.)

Example:

```
var smarties =new String("A generation of genius.");
var findIt = smarties.match(/genius/gi); //Gobal and ignore
case
if(findIt) {
document.write("The match is made!")
}
```

`string.replace(re,newStng)`

Replaces the contents of a regular expression with a new string. (Use Perl regular expression format.)

Example:

```
var smarties =new String("A generation of genius.");
var replaceIt = smarties.replace(/genius/,"science")
document.write(replaceIt) //Returns 'A generation of science.'
```

`string.search(re)`

Searches for a regular expression and returns the starting point.

Example:

```
var smarties =new String("A generation of genius.");
var searchIt = smarties.search(/rat/)
document.write("Mr. Rat begins at position " + searchIt +".")
```

`string.slice(b,e)`

Creates a substring beginning at position `b` and ending at `e`.

Example:

```
var pet = new String("Greater Swiss Mountain Dog")
var alpine=pet.slice(14,22);
document.write(alpine)
```

`string.small( )`

Creates a font in the format of `<small>` tag in HTML.

Example:

```
var littleGuy=new String("Chihuahua");
document.write(littleGuy.small( ));
```

`string.split(d)`

Creates an array of strings from a single string, using a delimiter (`d`) to break the string into elements.

Example:

```
var farmersMarket=new String("strawberries-cantelope-oranges-
apples")
var order=farmersMarket.split("-");
document.write(order[3]);
```

`string.strike( )`

Creates strikethrough characters. Based on the `<strike>` tag in HTML.

Example:

```
var alpha=new String("Trash");
document.write(alpha.strike( ));
```

`string.sub( )`

Creates a subscript font. Based on the `<sub>` tag in HTML.

Example:

```
var alpha=new String("Australia");
document.write("That country is down under " + alpha.sub( ));
```

`string.substr (b,l)`

A substring of a string beginning at `b` and a length of `l`.

(This is a subtle but important difference from `string.substring( )`.)

Example:

```
var alpha=new String("JavaScript is too cool.");
var work=alpha.substr(4,6) +"ing can be hard work."
document.write(work);
```

`string.substring(b,e)`

A substring of a string beginning at `b` and ending at `e`.

Example:

```
var alpha=new String("JavaScript is too cool.");
var work=alpha.substring(4,9) +"ing can be hard work."
document.write(work);
```

`string.sup( )`

Creates a superscript font position. Based on the `<sup>` tag in HTML.

Example:

```
var alpha=new String("Arctic Circle");
document.write("That place is way up there " + alpha.sup( ));
```

`string.toLowerCase( )`

Forces all characters in a string to lowercase.

Example:

```
var alpha=new String("ALL UPPER CASE");
document.write(alpha.toLowerCase( ));
```

`string.toUpperCase( )`

Forces all characters in a string to lowercase.

Example:

```
var alpha=new String("all lower case");
document.write(alpha.toUpperCase());
```

`unescape( )`

Changes characters from escape format to decoded format.

Example:

```
var alpha=new String("Testing%20one");
document.write(unescape(alpha));
```

`window`

Reference to window object.

Example:

```
window
```

`window.alert( )`

See [alert(value)]

`window.clearInterval( )`

Stops actions initiated by `setInterval( )`.

Example:

```
window.clearInterval.
```

`window.clearTimeout( )`

Cancels `setTimeout( )`.

Example:

```
if (var clocker==4) {
window.clearTimeout( )
}
```

`window.close( )`

Closes a specified window or the current window. (*See* `window.open( )`.)

Example:

```
window.close( );
ralph.close( );//ralph is variable name defined in opening.
```

`window.closed`

Tests whether a specified window has been closed.

Example:

```
var checkWin=smWin.closed; //smWin is a var name defined
//when window was opened
if(checkWin) {....
```

`window.confirm(q)`

Presents a question to ask the user. A cancel returns `false`.

Example:

```
function checkFirst( ) {
          if(window.confirm("You really want to close it?")) {;
                window.close( );
          }
}
```

`window.defaultStatus`

A read/write string property used to display a message in the window status line.

Example:

```
var message="Look down here!";
window.defaultStatus=message;
//The string "Look down here!" appears in status line.
```

`window.document`

Reference to the current document. (The window term is usually superfluous.)

Example:

```
var alpha=window.document.forms[2].elements[7].value;
```

`window.focus( )`

Provides keyboard focus to a specified window or frame.

Example:

```
windows.frames[2].focus( )
```

`window.frames[ ]`

Reference to frames within a window.

Example:

```
var numFrames=window.frames.length;
```

`window.length`

Returns the number of frames in a window.

Example:

```
var numFrames=window.length;
```

`window.location`

URL of the current HTML page loaded.

Example:

```
var where=window.location;
```

See also [**location**]

`window.moveBy(rx,ry)`

The number of pixels to move the window to the right and down. (NN requires `UniversalBrowserWrite` privilege to move off the screen.)

Example:

```
window.moveBy(40,50)
```

`window.moveTo(ax,ay)`

Moves a window to absolute x,y position. (NN requires `UniversalBrowserWrite` privilege to move off the screen.)

Example:

```
window.moveTo(250,400)
```

`window.name`

Name of the window specified in the `window.open( )` statement.

Example:

```
var louise=window.open("","flowers")
//"flowers" is the window's name, but to close the window use,
//louise.close( )
```

`window.open( )`

Opens a new window for an existing page or new page. May use variable definition to open a window by naming arguments (`url, name, features`, and `replace`). Both major browsers accept the following features:

```
height
location
menubar
resizable
scrollbars
status
toolbar
width
```

All arguments are separated by commas, and features are in parentheses and separated by spaces.

Example:

```
var winName=window.open(" ","services","height=200
width=300",true)
```

`window.parent`

A frame’s parent. Usually used in a frame page’s script to reference another frame.

Example:

```
parent.side.document.location="http://www.sandlight.com"
//'side' is a frame name in the same parent window.
```

`window.prompt(q,d)`

A prompt window appears with question `q` and optional default `d`. User feedback can go into a variable.

Example:

```
function promptMe( ) {
var alpha=window.prompt("How old are you",18);
document.forms[0].elements[1].value=alpha;
}//The variable alpha contains whatever the user typed in.
```

`window.resizeBy(rh,rw)`

Resizes the window by `rh` height in pixels and `rw` width. Function adds pixels to current size. (NN requires `UniversalBrowserWrite` privilege to set either `ah` or `aw` to less than 100 pixels.)

Example:

```
window.resizeBy(200,100)
```

`window.resizeTo(ah,aw)`

Resizes the window to the absolute height and width specified. (NN requires `UniversalBrowserWrite` privilege to set either `ah` or `aw` to less than 100 pixels.)

Example:

```
window.resizeTo(580,400)
```

`window.screen`

See [**window.screen**]

`window.scrollTo(x,y)`

Scrolls page to x,y coordinates on screen.

Example:

```
scrollTo(300,200);
```

`window.self`

Used mainly to clarify a window reference to itself.

Example:

```
window.self
```

`window.setInterval(script,milliseconds)`

Executes a script at an interval set in milliseconds. Both NN and IE use this form. IE does not use the second form, `window.setInterval(function,milliseconds, arguments)`.

Example:

```
window.setInterval(alpha += 3, 5000);
```

`window.setTmeout(f,d)`

Delays execution (`f`) for the specified number of milliseconds (`d`).

Example:

```
function ready( ) {
window.setTimeout("document.reactR.src=react4.src",500);
window.setTimeout("document.reactR.src=react1.src",1000);
}
```

`window.status`

A read/write string property used to read or add transient message to the status line.

Example:

```
var temp = "Your frame now is Pictures."
window.status = temp;
```

`window.top`

Usually used with frames to reference the top-level window in the frame.

Example:

```
var topDog=window.top;
```

`window.window`

See [**window.self**]