

SYBEX Bonus Chapter

HTML Complete

Sybx

Bonus Chapter: Introducing JavaScript and JScript

Copyright © 2003 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-4209-5

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other countries.

TRADEMARKS: Sybx has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.

This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Sybx Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybx.com

BONUS CHAPTER: INTRODUCING JAVASCRIPT AND JSRIPT

Adapted from *Mastering JavaScript*, by James Jaworski
ISBN 0-7821-2819-X
\$49.99

This chapter introduces you to the JavaScript language. I'll show you how JavaScript works with both the Netscape and Microsoft browsers and web servers and how to embed JavaScript statements in HTML and XHTML documents. Then I'll cover JavaScript's use of *types* and *variables* and show you how to use *arrays*. By the time you finish this chapter, you'll be able to write simple scripts and include them in your web pages.

JavaScript with Browsers and Servers

JavaScript is a script-based programming language that supports the development of both client and server components of web-based applications. On the client side, it can be used to write programs that are executed by a web browser within the context of a web page. On the server side, it can be used to write web server programs that can process information submitted by a web browser and then update the browser's display accordingly.

& Microsoft's version of JavaScript is named JScript. I use "JavaScript" to refer to both JavaScript and JScript unless I'm referring to one but not the other. In these cases, I'll refer to "Netscape's JavaScript" and "Microsoft's JScript."

As you know, a web browser displays a web page by acting on the instructions contained in an HTML/XHTML file. The browser reads the HTML/XHTML file and displays elements of the file as they are encountered. In this case, the HTML/XHTML file (which the browser has retrieved from a web server, seen on the right) contains embedded JavaScript code. The process of reading the HTML/XHTML file and identifying the elements contained in the file is referred to as *parsing*. When a script is encountered during parsing, the browser executes the script before continuing with further parsing.

The script can perform actions, such as generating HTML/XHTML code that affects the display of the browser window. It can perform actions that affect the operation of plug-

ins, Java applets, or ActiveX components. The script also can define JavaScript language elements that are used by other scripts.

Some scripts may define functions for handling *events* that are generated by user actions. For example, you might write a script to define a function for handling the event “submitting a form” or “clicking a link.” The event handlers then can perform actions such as validating the form’s data, generating a custom URL for the link, or loading a new web page.

JavaScript’s event-handling capabilities provide greater control over the user interface than HTML or XHTML alone. For example, when a user submits an HTML form, a browser that isn’t implementing JavaScript handles the “submit form” event by sending the form data to a CGI program for further processing. The CGI program processes the form data and returns the results to the web browser, which displays the results to the user. By comparison, when a user submits an HTML form using a browser that *does* implement JavaScript, a JavaScript event-handling function may be called to process the form data. This processing may vary from validating the data (that is, checking to see that the data entered by the user is appropriate for the fields contained in the form) to performing all of the required form processing, eliminating the need for a CGI program. In other words, JavaScript’s event-handling capabilities allow the *browser* to perform some, if not all, of the form processing. Besides providing greater control over the user interface, these event-handling capabilities help to reduce network traffic, the need for CGI programs, and the load on the web server.

While JavaScript’s browser programming capabilities can eliminate the need for *some* server-side programs, others still are required to support more advanced web applications, such as those that access database information, support electronic commerce, or perform specialized processing. Server-side JavaScript scripts are used to replace traditional CGI programs. Instead of a web server calling a CGI program to process form data, perform searches, or implement customized web applications, a JavaScript-enabled web server can invoke a precompiled JavaScript script to perform this processing. The web server automatically creates JavaScript objects that tell the script how it was invoked and the type of browser requesting its services. It also automatically communicates any data supplied by the browser. The script processes the data provided by the browser and returns information to the browser via the server. The browser then uses this information to update the user’s display.

There are several advantages to using server-side JavaScript scripts on Netscape and Microsoft web servers:

- Because these web servers have been specially designed for executing JavaScript scripts, they are able to minimize the processing overhead that is usually associated with invoking the script, passing data, and returning the results of script processing.
- You can use JavaScript to replace CGI scripts written in other languages. This eliminates the problems that usually are associated with managing multiple CGI programs, which may have been written in an OS shell language, Perl, Tcl, C,

and other languages. It also provides tighter control over the security of those server-side applications.

- The database extensions integrated within those servers provide a powerful capability for accessing information contained in compatible external databases. These database extensions may be used by server-side scripts.

The database connectivity supported by these servers enables even beginning programmers to create server-side JavaScript programs to update databases with information provided by browsers (usually through forms) and to provide web users with web pages that are generated dynamically from database queries. You can imagine how exciting this is for researchers gathering and reporting information over the web and for entrepreneurs who have catalogs full of products and services to sell over the web.

& In this section, I've provided an overview of the different ways in which JavaScript can be used for browser and server-side web applications. JavaScript's syntax is the same for both client (browser) *and* server programming; however, the examples I will be using in this chapter mainly reflect how JavaScript relates to browser programming.

Embedding JavaScript in HTML/XHTML

JavaScript statements can be included in HTML/XHTML documents by enclosing the statements between an opening `<script>` tag and a closing `</script>` tag. Within the opening tag, the `language` attribute is set to "JavaScript" to identify the script as being JavaScript as opposed to some other scripting language, such as Visual Basic Script (VBScript). The script tag is typically used as follows:

```
<script language="JavaScript" type="text/javascript">  
  JavaScript statements  
</script>
```

The script tag may be placed in either the head or the body of an HTML/XHTML document. In many cases, it is better to place the script tag in the head of a document to ensure that all JavaScript definitions have been made before the body of the document is displayed. You'll learn more about this in the section "Use of the Document Head," later in this chapter.

& This code includes a `type` attribute with the value `text/javascript`. The required `type` attribute in the `script` element is used to specify a Multipurpose Internet Mail Extensions (MIME) type for the scripting language. The HTML 4.01 specification recommends `text/javascript` as the MIME type for JavaScript. However, this type has not been registered with the Internet Assigned Numbers Authority (IANA) as an official MIME type. Another MIME type, `application/x-javascript`, has been registered with IANA. However, to further

complicate this issue, in Internet Explorer 4.0 and above, in order to run a script using JavaScript, you must use a MIME type of `text/javascript`, `text/jscript`, or `text/ecmascript`. For more information on scripting and the `type` attribute, see www.webreference.com/html/tutorial16/4.html.

The traditional first exercise with any programming language is to write a program to display the text Hello World! This teaches the programmer to display output, a necessary feature of most programs. A JavaScript script that displays this text is shown in Listing 22.1.

Listing 22.1: Hello World!

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Hello World!</title>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<script language="javascript" type="text/javascript">
document.write("Hello World!")
</script>
</body>
</html>
```

The body of the example document (the lines between the `<body>` and the `</body>` tags) contains a single element: a script, identified by the `<script>` and `</script>` tags. The opening script tag has the attribute `language="javascript"` to identify the script as JavaScript and the attribute `type="text/javascript"` to specify the required script MIME type. The script has a single statement, `document.write("Hello World!")`, that writes the text Hello World! to the body of the current document object.

Other Language Attributes

All JavaScript-capable browsers will process JavaScript code if the `language` attribute is set to `javascript`. However, the `language` attribute also can be set to the following other values in order to limit the browsers that are able to process JavaScript code.

javascript1.1

This is used to limit execution of a script to browsers that support JavaScript 1.1. Those browsers are Navigator 3 and later, Internet Explorer 4 and later, HotJava 2.0 and later, and Opera 3.5 and later.

javascript1.2

This is used to limit execution of a script to browsers that support JavaScript 1.2. Those browsers are Navigator 4 and later, Internet Explorer 4 and later, Opera 4 and later, and HotJava 3.0 and later.

javascript1.3

This is used to limit execution of a script to browsers that support JavaScript 1.3. Those browsers are Navigator 4.06 and later, Internet Explorer 5 and later, Opera 5 and later, and HotJava 3.0 and later.

javascript1.5

This is used to limit execution of a script to browsers that support JavaScript 1.5. Those browsers are Navigator 6.0 and later, Internet Explorer 5.5 and later, and Opera 7 and later.

jscript

This is used to limit execution of a script to browsers that support JScript. Those browsers are limited to Internet Explorer 3 and later. The Opera browser may be configured to support JScript by setting it to Internet Explorer mode. Table 22.1 identifies which of the attributes listed are supported by popular browsers. If a browser does not support an attribute, it will simply ignore the <script> tags.

Table 22.1: Browser Support of the language Attribute

BROWSER	JAVASCRIPT	JAVASCRIPT PT 1.1	JAVASCRIPT PT 1.2	JAVASCRIPT PT 1.3	JAVASCRIPT PT 1.5	JSCRIPT
Navigator 4	X	X	X			
Navigator 4.06	X	X	X	X		
Navigator 6.0, 6.1, and later	X	X	X	X	X	
Internet Explorer 4	X	X	X			X
Internet	X	X	X	X		X

Explorer 5						
Internet Explorer 5.5	X	X	X	X	X	X
Internet Explorer 6 and later		X	X	X	X	X
Opera 4	X	X	X	X		X
Opera 5 and 6	X	X	X	X		
Opera 7	X	X	X	X	X	
HotJava 3.0	X	X	X	X		

You may wonder whatever happened to JavaScript 1.4? JavaScript 1.4 corresponds to ECMAScript Revision 2. The only browsers that recognize and support the `javascript1.4` attribute value are HotJava 3.0 and Opera 5 and later.

F To ensure that more browsers are able to execute your scripts, set the `language` attribute to `javascript`. Your JavaScript code then can perform checks to detect which type and version of browser currently is executing a script.

& In addition to the `language` attribute, Internet Explorer 4 and later support the use of conditional compilation directives. These directives are used to limit script execution to selected portions of scripts.

Telling Non-JavaScript Browsers to Ignore Your Code

Not all browsers support JavaScript. Older browsers, such as Netscape Navigator 1, Internet Explorer 2, and the character-based Lynx browser, do not recognize the script tag and, as a consequence, display as text all the JavaScript statements that are enclosed between `<script>` and `</script>`

Fortunately, HTML/XHTML provides a method to conceal JavaScript statements from such JavaScript-challenged browsers. The trick is to use HTML/XHTML comment tags to surround the JavaScript statements. Because HTML/XHTML comments are displayed only within the code used to create a web page, they do not show up as part of the browser's display. The use of HTML/XHTML comment tags is as follows:

```
<!-- Begin hiding JavaScript
JavaScript statements
// End hiding JavaScript -->
```

The `<!--` tag begins the HTML/XHTML comment, and the `-->` tag ends the comment. The `//` string identifies a JavaScript comment, as you'll learn later in this chapter in the "JavaScript Comments" section.

The comment tags cause the JavaScript statements to be treated as comments by JavaScript-challenged browsers. JavaScript-enabled browsers, on the other hand, know to ignore the comment tags and process the enclosed statements as JavaScript. Listing 22.2 shows how XHTML comments are used to hide JavaScript statements.

Listing 22.2: Using HTML Comments to Hide JavaScript Code

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using HTML comments to hide JavaScript code</title>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<script language="JavaScript" type="text/javascript">
<!-- Begin hiding JavaScript
document.write("Hello World!")
// End hiding JavaScript -->
</script>
</body>
</html>
```

The *noscript* Tag

Version 2 and later of Netscape Navigator and version 3 and later of Microsoft Internet Explorer support JavaScript. These browsers account for nearly 98 percent of browser use on the web, and their percentage of use is increasing. This means that most browser requests come from JavaScript-capable browsers. However, there are still popular browsers such as Lynx that do not support JavaScript.

In addition, both Navigator and Internet Explorer provide users with the option of disabling JavaScript. The `noscript` tag was created for those browsers that can't or won't process JavaScript. It is used to display markup that is an alternative to executing a script. The HTML/XHTML instructions contained inside the tag are displayed by JavaScript-challenged browsers (as well as by JavaScript-capable browsers that have JavaScript disabled).

The script shown in Listing 22.3 illustrates the use of the `noscript` tag.

Listing 22.3: Using the `noscript` Tag

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using the noscript tag.</title>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
```

```
</head>
<body>
<script language="javascript" type="text/javascript">
<!-- Begin hiding JavaScript
document.write("Hello World!")
// End hiding Javascript -->
</script>
<noscript>
[JavaScript]
</noscript>
</body>
</html>
```

The *script* Element's *src* Attribute

The `script` element itself provides another way to include JavaScript code in an HTML/XHTML document, via the element's `src` attribute, which may be used to specify a file containing JavaScript statements. Here's an example of the use of the `src` attribute:

```
<script language="JavaScript" src="src.js"
  type="text/javascript">
</script>
```

In this example, `src.js` is a file containing JavaScript statements. (The file could have been named anything, but it should end with the `.js` extension; I chose `src.js` to help you remember the `src` attribute.) Note that the closing `</script>` tag still is required.

Listing 22.4: Inserting Source JavaScript Files

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using the src attribute of the script tag.</title>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<script language="JavaScript" src="src.js"
  type="text/javascript">
</script>
</body>
</html>
```

& The `src` attribute may have a URL as its attribute value. Web servers that provide the source file, however, *must* report the file's MIME type as `application/x-javascript`; otherwise, browsers will not load the source file.

JavaScript Comments

The JavaScript language provides comments of its own. Comments are used to insert notes and processing descriptions into scripts. The comments are ignored (as intended) when the statements of a script are parsed by JavaScript-enabled browsers.

JavaScript comments use the syntax of C++ and Java. The `//` string identifies a comment that continues to the end of a line. This is an example of a single line comment:

```
// This JavaScript comment continues to the end of the line.
```

The `/*` and `*/` strings are used to identify comments that span multiple lines. The comment begins with `/*` and continues up to `*/`. Here is an example of a multiple line comment:

```
/* This is
an example
of a multiple
line comment */
```

The script shown in Listing 22.5 illustrates the use of JavaScript comments. The script contains four statements that, if they weren't ignored, would write various capitalizations of the text Hello World! to the current document. However, because the first three of these statements are contained in comments, and because browsers ignore comments, these statements have no effect on the web page generated by the script.

Listing 22.5: Using JavaScript Comments

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using JavaScript comments</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>
<script language="JavaScript" type="text/javascript">
<!-- Begin hiding JavaScript
// document.write("hello world!")
/* document.write("Hello world!")
document.write("Hello World!") */
document.write("HELLO WORLD!")
// End hiding Javascript -->
</script>
</body>
</html>
```

<p>& Throughout the rest of the book, all browser references will be to JavaScript-capable browsers unless otherwise specified.</p>

Use of the Document Head

The head of an HTML/XHTML document provides a great place to include JavaScript definitions. Because the head of a document is processed before its body, placing definitions in the head will cause them to be defined before they are used. This is important because any attempt to use a variable before it is defined results in an error.

Listing 22.6 shows how JavaScript definitions can be placed in the head of an XHTML document. The script contained in the document head defines a variable named `greeting` and sets its value to the string `Hi Web surfers!` (You’ll learn all about variables in the section “Variables—Value Storehouses,” later in this chapter.) The script contained in the document’s body then writes the value of the `greeting` variable to the current document.

Listing 22.6: Using the Head for Definitions

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using the HEAD for definitions</title>
<script language="javascript" type="text/javascript">
<!--
greeting = "Hi Web surfers!"
// -->
</script>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<script language="javascript" type="text/javascript">
<!--
document.write(greeting)
// -->
</script>
</body>
</html>
```

It is important to make sure that all definitions occur before they are used; otherwise an error will be displayed when your HTML document is loaded by a browser.

Listing 22.7 contains an XHTML document that will generate a “use before definition” error. In this listing, the head contains a JavaScript statement that writes the value of the `greeting` variable to the current document; however, the `greeting` variable is not defined until the body of the document.

Listing 22.7: Example of a Use Before Definition Error

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```

<title>Use before definition</title>
<script language="javascript" type="text/javascript">
<!--
document.write(greeting)
// -->
</script>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<script language="javascript" type="text/javascript">
<!--
greeting = "Hi Web surfers!"
// -->
</script>
</body>
</html>

```

Generating HTML/XHTML

The examples presented so far have shown how you can use JavaScript to write simple text to the `document` object. By including HTML/XHTML tags in your JavaScript script, you also can use JavaScript to generate HTML/XHTML elements that will be displayed in the current document. The example shown in Listing 22.8 illustrates this concept.

Listing 22.8: Using JavaScript to Create XHTML Tags

```

<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using JavaScript to create HTML tags</title>
<script language="javascript" type="text/javascript">
<!--
greeting = "<h1>Hi Web surfers!</h1>"
welcome = "<p>Welcome to <cite>Mastering JavaScript and
  JScript</cite>.</p>"
// -->
</script>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<script language="javascript" type="text/javascript">
<!--
document.write(greeting)
document.write(welcome)
// -->
</script>
</body>
</html>

```

In the script contained in the head of the XHTML document, the variables `greeting` and `welcome` are assigned text strings containing embedded XHTML tags. These text strings are displayed by the script contained in the body of the XHTML document:

- The `greeting` variable contains the heading `Hi Web surfers!`, which is surrounded by the XHTML heading tags `<h1>` and `</h1>`.
- The `welcome` variable is assigned the string `Welcome to Mastering JavaScript and JScript`.
- The citation tags, `<cite>` and `</cite>`, cause the `welcome` variable's string to be cited as a literary reference (which means it shows up in italic).
- The paragraph tags, `<p>` and `</p>`, which surround the `welcome` text, are used to mark it as a separate paragraph.

The resulting XHTML document generated by the script is equivalent to the following:

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using JavaScript to create XHTML tags</title>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<h1>Hi Web surfers!</h1>
<p>Welcome to <cite>Mastering JavaScript</cite>.</p>
</body>
</html>
```

So far, I've been using variables, such as `greeting` and `welcome`, without having explicitly defined what they are. The next section introduces variables.

Variables—Value Storehouses

JavaScript, like other programming languages, uses variables to store values to be used in other parts of a program. Variables are names that are associated with stored values. For example, the variable `imageName` may be used to refer to the name of an image file to be displayed, and the variable `totalAmount` may be used to display the total amount of a user's purchase.

Variable names can begin with an uppercase letter (A through Z), lowercase letter (a through z), an underscore character (`_`), or dollar sign character (`$`). The remaining characters can consist of letters, the underscore character, the dollar sign character, or digits (0 through 9). Examples of variable names are as follows:

- `orderNumber2`

- `_123`
- `SUM`
- `Image7`
- `Previous_Document`

Variable names are case sensitive. This means that a variable named `sum` refers to a different value than one named `Sum`, `sUm`, or `SUM`.

M Because variable names are case sensitive, it is important to make sure that you use the same capitalization each time you use a variable.

M The dollar sign (\$) character is reserved for machine-generated code and should not be used in your scripts. In particular, it should not be used for scripts that will be run by earlier browsers that are not fully ECMAScript compatible.

Types and Variables

Unlike Java and some other programming languages, JavaScript does not require you to specify the *type* of data contained in a variable. (It doesn't even allow it.) In fact, the same variable may be used to contain a variety of different values, such as the text string `Hello World!`, the integer `13`, the floating-point value `3.14`, or the logical value `true`. The JavaScript interpreter keeps track of and converts the type of data contained in a variable.

JavaScript's automatic handling of different types of values is a double-edged sword. On one side, it frees you from having to explicitly specify the type of data contained in a variable and from having to convert from one data type to another. On the other side, because JavaScript automatically converts values of one type to another, it is important to keep track of what types of values should be contained in a variable and how they are converted in expressions involving variables of other types. The next section, "Types and Literal Values," identifies the types of values that JavaScript supports. The "Conversion Between Types" section later in this chapter covers important issues related to type conversion.

Types and Literal Values

JavaScript supports five primitive types of values. *Primitive* types are those that can be assigned a single literal value, such as a number, string, or Boolean value. JavaScript also supports complex types, such as arrays and objects. Here are the primitive types that JavaScript supports:

- **Number**—Consists of integer and floating-point numbers and the special NaN (not a number) value. Numbers use a 64-bit IEEE 754 format.
- **Boolean**—Consists of the logical values `true` and `false`.
- **String**—Consists of string values that are enclosed in single or double quotes.
- **Null**—Consists of a single value, `null`, which identifies a null, empty, or nonexistent reference.
- **Undefined**—Consists of a single value, `undefined`, which is used to indicate that a variable has not been assigned a value.

M The `undefined` value was introduced with the ECMAScript specification and is not supported by browsers that are not fully ECMAScript compatible. This includes Navigator 4.05 and earlier and Internet Explorer 3 and earlier.

& You'll learn about the Object type in the section "The Object Type and Arrays," later in this chapter.

In JavaScript, you do not declare the type of a variable as you do in other languages, such as Java and C++. Instead, the type of a variable is defined implicitly based on the literal value you assign to it. For example, if you assign the integer `123` to the variable `total`, then `total` will support number operations. If you assign the string value `The sum of all accounts` to `total`, then `total` will support string operations. Similarly, if you assign the logical value `true` to `total`, it will support Boolean operations.

It also is possible for a variable to be assigned a value of one type and then be assigned a value of another type later in the script's execution. For example, the variable `total` could be assigned `123`, then `The sum of all accounts`, and then `true`. The type of the variable would change with the type of value assigned to it. The different types of literal values that can be assigned to a variable are covered in the following sections.

Number Types—Integers and Floating-Point Numbers

When working with numbers, JavaScript supports both integer and floating-point values. It transparently converts from one type to another as values of one type are combined with values of other types in numerical expressions. For example, integer values are converted to floating-point values when they are used in floating-point expressions.

Integer Literals

Integers can be represented in JavaScript in decimal, hexadecimal, or octal form:

- A *decimal* (base-10) integer is what nonprogrammers are used to seeing—the digits 0 through 9, with each new column representing a higher power of 10.

- A *hexadecimal* (base-16) integer in JavaScript must always begin with the characters 0x or 0X in the two leftmost columns. Hexadecimal uses the numbers 0 through 9 to represent the values zero through nine and the letters A through F to represent the values that nonprogrammers know as 10 through 15.
- An *octal* (base-8) integer in JavaScript must always begin with the character 0 (zero) in the leftmost column. Octal uses only the digits 0 through 7.

Examples of decimal, hexadecimal, and octal integers are provided in Table 22.2.

Table 22.2. Examples of Decimal, Hexadecimal, and Octal Integers for the Same Values

DECIMAL NUMBER	HEXADECIMAL EQUIVALENT	OCTAL EQUIVALENT
19	0x13	023
255	0xff	0377
513	0x201	01001
1024	0x400	02000
12345	0x3039	030071

The program shown in Listing 22.9 illustrates the use of JavaScript hexadecimal and octal integers.

Listing 22.9: Using JavaScript Integers

```
<!DOCTYPE html PUBLIC
-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using JavaScript integers</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>
<script language="javascript" type="text/javascript">
<!--
document.write("0xab00 + 0xcd = ")
document.write(0xab00 + 0xcd)
document.write("<BR>")
document.write("0xff - 0123 = ")
document.write(0xff - 0123)
document.write("<BR>")
document.write("-0x12 = ")
document.write(-0x12)
// -->
</script>
</body>
</html>
```

Floating-Point Literals

Floating-point literals are used to represent numbers that require the use of a decimal point, or very large or small numbers that must be written using exponential notation.

A floating-point number must consist of either a number containing a decimal point or an integer followed by an exponent. The following are valid floating-point numbers:

```
-4.321
55.
12e2
1e-2
7e1
-4e-4
.5
```

As you can see in these examples, floating-point literals may contain an initial integer, followed by an optional decimal point and fraction, followed by an optional exponent (e or E) and its integer exponent value. For example, $4e6$ equals 4×10 to the sixth power, which equals 4,000,000. Also, the initial integer and integer exponent value may be signed as positive or negative (+ or -). Up to 20 significant digits may be used to represent floating-point values.

The script shown in Listing 22.10 illustrates how JavaScript displays these values. Notice that JavaScript simplifies the display of these numbers whenever possible.

Listing 22.10: Using Floating-Point Numbers

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using floating-point numbers</title>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<script language="javascript" type="text/javascript">
<!--
document.write(-4.321)
document.write("<br />")
document.write(55.)
document.write("<br />")
document.write(12e2)
document.write("<br />")
document.write(1e-2)
document.write("<br />")
document.write(7e1)
document.write("<br />")
```

```
document.write(-4e-4)
document.write("<br />")
document.write(.5)
// -->
</script>
</body>
</html>
```

Boolean Values

A *Boolean value* is a value that is either true or false. The word *Boolean* is taken from the name of the English mathematician George Boole, who developed much of the fundamental theory of mathematical logic.

JavaScript, like Java, supports a pure Boolean type that consists of the values `true` and `false`. Several logical operators may be used in Boolean expressions. JavaScript automatically converts the Boolean values `true` and `false` into 1 and 0 when they are used in numerical expressions. The script shown in Listing 22.11 illustrates this automatic conversion.

Listing 22.11: Conversion of Logical Values to Numeric Values

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Conversion of logical values to numeric values
</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>
<script language="javascript" type="text/javascript">
<!--
document.write("true*5 + false*7 = ")
document.write(true*5 +false*7)
// -->
</script>
</body>
</html>
```

String Values

JavaScript provides built-in support for strings of characters. A *string* is a sequence of zero or more characters that are enclosed by double quotes (") or single quotes ('). If a string begins with a double quote, it must end with a double quote. Likewise, if a string begins with a single quote, it must end in a single quote.

To insert a single or double quote character in a string, you must precede it by the backslash escape character (\). The following are examples of the use of the escape character to insert quotes into strings:

```
"He asked, \"Who owns this book?\""
'It\'s Bill\'s book.'
```

The script shown in Listing 22.12 illustrates the use of quotes within strings. Note that single quotes do not need to be coded with escape characters when they are used within double-quoted strings. Similarly, double quotes do not need to be coded when they are used within single-quoted strings.

Listing 22.12: Using Quotes within Strings

```
<!DOCTYPE html PUBLIC
-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using quotes within strings</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>
<script language="javascript" type="text/javascript">
<!--
document.write("He said, \"That's mine!\"<BR>")
document.write('She said, "No it\'s not."<BR>')
document.write('That\'s all folks!')
// -->
</script>
</body>
</html>
```

JavaScript defines special formatting characters for use in strings. These characters are identified in Table 22.3.

Table 22.3. Special Formatting Characters

CHARACTER	MEANING
\'	Single quote
\"	Double quote
\\	Backslash
\n	Newline
\r	Carriage return
\f	Formfeed
\t	Horizontal tab
\b	Backspace
\v	Vertical tab

The script shown in Listing 22.13 shows how these formatting characters are used. The web page uses the XHTML *preformatted text* tags to prevent the formatting characters from being treated as XHTML white space characters. Notice that the backspace character and the formfeed character are displayed incorrectly and that the carriage return character is displayed in the same manner as the newline character. Even though these characters are not fully supported in the display of web pages, they still may be used to insert formatting

codes within data and files that JavaScript produces. The way that text is displayed is also dependent on how the user has configured the browser's default font and language settings.

& Any Unicode character may be encoded using a special escape sequence consisting of `\uxxxx`, where each `x` is a hexadecimal digit and the four digits provide the Unicode value for the character. For example, `\u0041` is the escape sequence for the letter `A`.

Listing 22.13: Using Special Formatting Characters

```
<!DOCTYPE html PUBLIC
-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using special formatting characters</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>
<pre>
<script language="javascript" type="text/javascript">
<!--
document.write("This shows how the \backspace character
works.\n")
document.write("This shows how the \tab character
works.\n")
document.write("This shows how the \r carriage return
character works.\n")
document.write("This shows how the \f form feed character
works.\n")
document.write("This shows how the \n new line character
works.\n")
// -->
</script>
</pre>
</body>
</html>
```

The *null* Value

The `null` value is common to all JavaScript types. It is used to set a variable to an initial value that is different from other valid values. Use of the `null` value prevents the sort of errors that result from using uninitialized variables. The `null` value automatically is converted to default values of other types when used in an expression, as you'll see shortly in the "Conversion Between Types" section.

The *undefined* Value

The *undefined* value indicates that a variable has been created but not assigned a value. Like the `null` value, the *undefined* value is common to all JavaScript types and is automatically converted to default values of these types. The *undefined* value is converted to `NaN` for numeric types, `false` for Boolean, and *undefined* for strings.

Conversion Between Types

JavaScript automatically converts values from one type to another when they are used in an expression. This means that you can combine different types in an expression, and JavaScript will try to perform the type conversions that are necessary for the expression to make sense. For example, the expression, `"test" + 5` will convert the numeric 5 to a string 5 and append it to the string `test`, producing `test5`. JavaScript's automatic type conversion also allows you to assign a value of one type to a variable and then assign a value of a different type to the same variable later.

How does JavaScript convert from one type to another? The process of determining when a conversion should occur and what type of conversion should be made is fairly complex. JavaScript converts values when it evaluates an expression or assigns a value to a variable. When JavaScript assigns a value to a variable, it changes the type associated with the variable to the type of the value that is assigned.

When JavaScript evaluates an expression, it parses the expression into its component unary and binary expressions based on the order of precedence of the operators it contains. It then evaluates the component unary and binary expressions of the parse tree. Each expression is evaluated according to the operators involved. If an operator takes a value of a type that is different than the type of an operand, then the operand is converted to a type that is valid for the operator.

Some operators, such as the `+` operator, may be used for more than one type. For example, `"a"+"b"` results in the string `"ab"` when the `+` operator is used with string values, but it assumes its typical arithmetic meaning when used with numeric operands. What happens when JavaScript attempts to evaluate `"a"+3`? JavaScript converts the integer 3 into the string `"3"` and yields `"a3"` for the expression. In general, JavaScript will favor string operators over all others, followed by floating-point, integer, and logical operators.

The script shown in Listing 22.14 illustrates JavaScript conversion between types when the `+` operator is used.

Listing 22.14: Automatic Conversion Between Types

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Implicit conversion between types</title>
```

```

<script language="javascript" type="text/javascript">
<!--
s1 = "test"
s2 = "12.34"
i  = 123
r  = .123
lt = true
lf = false
n  = null
// -->
</script>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<h1>Implicit conversion between types</h1>
<table border="2">
<script language="javascript" type="text/javascript">
<!--
// Column headings for table
document.write("<tr>")
document.write("<th>row + column</th>")
document.write("<th>string \"12.34\"</th>")
document.write("<th>integer 123</th>")
document.write("<th>float .123</th>")
document.write("<th>logical true</th>")
document.write("<th>logical false</th>")
document.write("<th>>null</th>")
document.write("</tr>")
// First operand is a string
document.write("<tr>")
document.write("<th>string \"test\"</th>")
document.write("<td>"+(s1+s2)+"</td>")
document.write("<td>"+(s1+i)+"</td>")
document.write("<td>"+(s1+r)+"</td>")
document.write("<td>"+(s1+lt)+"</td>")
document.write("<td>"+(s1+lf)+"</td>")
document.write("<td>"+(s1+n)+"</td>")
document.write("</tr>")
// First operand is an integer
document.write("<tr>")
document.write("<th>integer 123</th>")
document.write("<td>"+(i+s2)+"</td>")
document.write("<td>"+(i+i)+"</td>")
document.write("<td>"+(i+r)+"</td>")
document.write("<td>"+(i+lt)+"</td>")
document.write("<td>"+(i+lf)+"</td>")
document.write("<td>"+(i+n)+"</td>")
document.write("</tr>")
// First operand is a float
document.write("<tr>")
document.write("<th>float .123</th>")
document.write("<td>"+(r+s2)+"</td>")
document.write("<td>"+(r+i)+"</td>")
document.write("<td>"+(r+r)+"</td>")
document.write("<td>"+(r+lt)+"</td>")
document.write("<td>"+(r+lf)+"</td>")
document.write("<td>"+(r+n)+"</td>")

```

```

document.write("</tr>")
// First operand is a logical true
document.write("<tr>")
document.write("<th>logical true</th>")
document.write("<td>"+(1t+s2)+"</td>")
document.write("<td>"+(1t+i)+"</td>")
document.write("<td>"+(1t+r)+"</td>")
document.write("<td>"+(1t+1t)+"</td>")
document.write("<td>"+(1t+1f)+"</td>")
document.write("<td>"+(1t+n)+"</td>")
document.write("</tr>")
// First operand is a logical false
document.write("<tr>")
document.write("<th>logical false</th>")
document.write("<td>"+(1f+s2)+"</td>")
document.write("<td>"+(1f+i)+"</td>")
document.write("<td>"+(1f+r)+"</td>")
document.write("<td>"+(1f+1t)+"</td>")
document.write("<td>"+(1f+1f)+"</td>")
document.write("<td>"+(1f+n)+"</td>")
document.write("</tr>")
// First operand is null
document.write("<tr>")
document.write("<th>null</th>")
document.write("<td>"+(n+s2)+"</td>")
document.write("<td>"+(n+i)+"</td>")
document.write("<td>"+(n+r)+"</td>")
document.write("<td>"+(n+1t)+"</td>")
document.write("<td>"+(n+1f)+"</td>")
document.write("<td>"+(n+n)+"</td>")
document.write("</tr>")
// -->
</script>
</table>
</body>
</html>

```

Note that in all cases where string operands are used with a non-string operand, JavaScript converts the other operand into a string:

- Numeric values are converted to their appropriate string value.
- Boolean values are converted to 1 and 0 to support numerical operations.
- The null value is converted to "null" for string operations, false for logical operations, and 0 for numerical operations.

Let's take a look at Listing 22.14, shown earlier. The script in the document head defines the variables to be used in the table's operations. The s1 and s2 variables are assigned string values. The i and r variables are assigned integer and floating-point values. The 1t and 1f variables are assigned logical values. The n variable is assigned the null value.

The script in the document body is fairly long. However, most of the script is used to generate the XHTML tags for the cells of the conversion table. The script is surrounded by the tags <table border="2"> and </table>. The script then generates the cells of the table one row at a time. The <tr> and </tr> tags mark a row of the table. The

<th> and </th> tags mark header cells. The <td> and </td> tags identify normal non-header table cells.

First, the column header row is displayed. Then each row of the table is generated by combining the operand at the row heading with the operand at the table heading using the + operator.

Conversion Functions

A *function* is a collection of JavaScript code that performs a particular task and often returns a value. A function may take zero or more parameters. The parameters are used to specify the data to be processed by the function.

JavaScript provides three functions that are used to perform explicit type conversion. They are `eval()`, `parseInt()`, and `parseFloat()`.

& A function is referenced by its name with the empty parameter list—()—appended. This makes it easier to differentiate between functions and variables in the discussion of scripts.

The `eval()` function can be used to convert a string expression to a numeric value. For example, the statement `total = eval("432.1*10")` results in the value 4321 being assigned to the `total` variable. The `eval()` function takes the string value "432.1*10" as a parameter and returns the numeric value 4321 as the result of the function call. When using the `eval()` function to convert a string expression to a numeric value, if the string value passed as a parameter to the `eval()` function does not represent a numeric value, then use of `eval()` results in an error being generated.

The `parseInt()` function is used to convert a string value into an integer. Unlike `eval()`, `parseInt()` returns the first integers contained in the string or 0 if the string does not begin with an integer. For example, `parseInt("123xyz")` returns 123, and `parseInt("xyz")` returns 0. The `parseInt()` function also parses hexadecimal and decimal integers.

The `parseFloat()` function is similar to the `parseInt()` function. It returns the first floating-point numbers contained in a string or 0 if the string does not begin with a valid floating-point number. For example, `parseFloat("2.1e4xyz")` returns 21000, and `parseFloat("xyz")` returns 0.

The script shown in Listing 22.15 illustrates the use of JavaScript's explicit conversion functions.

Listing 22.15: Explicit Conversion Functions

```
<!DOCTYPE html PUBLIC
-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```
<title>Using Explicit Conversion Functions</title>
<meta http-equiv="Content-Type" content="text/html;
  charset=iso-8859-1" />
</head>
<body>
<h1 align="center">Using Explicit Conversion Functions</h1>
<script language="javascript" type="text/javascript"><!--
document.write('eval("12.34*10") = ')
document.write(eval("12.34*10"))
document.write("<br />")
document.write('parseInt("0x10") = ')
document.write(parseInt("0x10"))
document.write("<br />")
document.write('parseFloat("5.4321e6") = ')
document.write(parseFloat("5.4321e6"))
// -->
</script>
</body>
</html>
```

The Object Type and Arrays

In addition to the primitive types discussed in the previous sections, JavaScript supports the Object type. This type is referred to as a complex data type because it is built from the primitive types. In this chapter, I'll cover a special JavaScript object—the array.

& An array is a special type of JavaScript object.

Arrays—Accessing Indexed Values

An *array* is an object that is capable of storing a sequence of values. The values are stored in indexed locations within the array. For example, suppose you have a company with five employees, and you want to display the names of your employees on a web page. You could keep track of their names in an array variable named `employee`. You would construct the array using the following statement:

```
employee = new Array(5)
```

You would store the names of your employees in the array using the following statements:

```
employee[0] = "Bill"
employee[1] = "Bob"
employee[2] = "Ted"
employee[3] = "Alice"
employee[4] = "Sue"
```

You could then access the names of the individual employees by referring to the individual elements of the array. For example, you could display the names of your employees using statements such as the following:

```
document.write(employee[0])
document.write(employee[1])
```

```
document.write(employee[2])
document.write(employee[3])
document.write(employee[4])
```

The script shown in Listing 22.16 illustrates the use of arrays.

Listing 22.16: Using JavaScript Arrays

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using Arrays</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>
<h1 align="center">Using Arrays</h1>
<script language="javascript" type="text/javascript"><!--
employee = new Array(5)
employee[0] = "Bill"
employee[1] = "Bob"
employee[2] = "Ted"
employee[3] = "Alice"
employee[4] = "Sue"
document.write(employee[0]+"<br />")
document.write(employee[1]+"<br />")
document.write(employee[2]+"<br />")
document.write(employee[3]+"<br />")
document.write(employee[4])
// --></script>
</body>
</html>
```

The *length* of an array is the number of elements it contains. In the example script of Listing 22.16, the length of the `employee` array is 5. The individual elements of an array are referenced using the name of the array followed by the index of the array element enclosed in brackets. Because the first index is 0, the last index is one less than the length of the array. For example, suppose that you have an array named `day` that has a length of 7 and contains the names of the days of the week. The individual elements of this array would be accessed as `day[0]`, `day[1]`, ..., `day[6]`.

Constructing Arrays

An array object must be constructed before it is used. An array may be constructed using either of the following two statement forms:

- `arrayName = new Array(arrayLength)`
- `arrayName = new Array()`

& A third form of array construction is discussed in the following section, "Constructing Dense Arrays."

In the first form, the length of the array is specified explicitly:

```
days = new Array(7)
```

In this example, `days` corresponds to the array name, and 7 corresponds to the array length.

In the second array construction form, the length of the array is not specified and results in the creation of an array of length 0:

```
order = new Array()
```

This constructs an array of length 0 that is used to keep track of customer orders. JavaScript automatically extends the length of an array when new array elements are initialized. For example, the following statements create an order array of length 0 and then subsequently extend the length of the array to 100 and then 1000.

```
order = new Array()  
order[99] = "Widget #457"  
order[999] = "Delux Widget Set #10"
```

When JavaScript encounters the reference to `order[99]` in this example, it extends the length of the array to 100 and initializes `order[99]` to "Widget #457". When JavaScript encounters the reference to `order[999]` in the third statement, it extends the length of `order` to 1000 and initializes `order[999]` to "Delux Widget Set #10".

Even if an array is created initially with a fixed length, it still may be extended by referencing elements that are outside the current size of the array. This is accomplished in the same manner as with zero-length arrays. Listing 22.17 shows how fixed-length arrays are expanded as new array elements are referenced.

Listing 22.17: Extending the Length of an Array

```
<!DOCTYPE html PUBLIC  
"-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml">  
<head>  
<title>Extending Arrays</title>  
<meta http-equiv="Content-Type" content="text/html;  
  charset=iso-8859-1" />  
</head>  
<body>  
<h1 align="center">Extending Arrays</h1>  
<script language="javascript" type="text/javascript"><!--  
order = new Array()  
document.write("order.length = "+order.length+"<br />")  
order[99] = "Widget #457"  
document.write("order.length = "+order.length+"<br />")  
order[999] = "Delux Widget Set #10"  
document.write("order.length = "+order.length+"<br />")  
// --></script>  
</body>  
</html>
```

Constructing Dense Arrays

A *dense array* is an array that is constructed initially with each element being assigned a specific value. Dense arrays are used in the same manner as other arrays, but they are constructed and initialized in a more efficient manner. Dense arrays are specified by listing the values of the array elements in place of the array length. Dense array declarations take the following form:

```
arrayName = new Array(value0, value1, ... , valuen)
Because we start counting at 0, the length of the array is n+1.
```

When you are creating short length arrays, the dense array declaration is very efficient. For example, an array containing the three-letter abbreviations for the days of the week may be constructed using this statement:

```
day = new Array('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat')
```

The Elements of an Array

JavaScript does not place any restrictions on the values of the elements of an array. These values could be of different types or could refer to other arrays or objects. For example, you could create an array as follows:

```
junk = new Array("s1", 's2', 4, 3.5, true, false, null,
    new Array(5, 6, 7))
```

The `junk` array has length 8, and its elements are as follows:

```
junk[0]="s1"
junk[1]='s2'
junk[2]=4
junk[3]=3.5
junk[4]=true
junk[5]=false
junk[6]=null
junk[7]=a new dense array consisting of the values 5, 6, & 7
```

The last element of the array, `junk[7]`, contains an array as its value. The three elements of `junk[7]` can be accessed using a second set of subscripts, as follows:

```
junk[7][0]=5
junk[7][1]=6
junk[7][2]=7
```

The script shown in Listing 22.18 illustrates the use of arrays within arrays.

Listing 22.18: An Array within an Array

```
<!DOCTYPE html PUBLIC
-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Arrays within Arrays</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>
```

```

<h1 align="center">Arrays within Arrays</h1>
<script language="javascript" type="text/javascript"><!--
junk = new Array("s1", 's2', 4, 3.5, true, false, null,
    new Array(5, 6, 7))
document.write("junk [0] = "+junk [0]+"<br />")
document.write("junk [1] = "+junk [1]+"<br />")
document.write("junk [2] = "+junk [2]+"<br />")
document.write("junk [3] = "+junk [3]+"<br />")
document.write("junk [4] = "+junk [4]+"<br />")
document.write("junk [5] = "+junk [5]+"<br />")
document.write("junk [6] = "+junk [6]+"<br />")
document.write("junk [7] [0] = "+junk [7] [0]+"<br />")
document.write("junk [7] [1] = "+junk [7] [1]+"<br />")
document.write("junk [7] [2] = "+junk [7] [2])
// -->
</script>
</body>
</html>

```

Objects and the *length* Property

JavaScript arrays are implemented as objects. An *object* is a named collection of data that has properties and may be accessed via methods. A *property* returns a value that identifies some aspect of the state of an object. *Methods* are used to read or modify the data contained in an object.

The length of an array is a property of an array. You can access the property of any object in JavaScript by appending a period (.) plus the name of the property to the name of the object, as shown here:

```
objectName.propertyName
```

For example, the length of an array is determined as follows:

```
arrayName.length
```

Now, consider the following array:

```
a = new Array(2, 4, 6, 8, 10)
```

The value returned by `a.length` is 5.

In addition to the `length` property, arrays also support several methods, including these:

- `concat()` combines two arrays into one new array.
- `join()` combines all elements in an array into a string.
- `reverse()` reverses the order of the elements in an array—the first element becomes the last, and the last becomes the first.
- `slice()` removes a section of an array.
- `sort()` sorts the array elements.

For more details on using JavaScript arrays and array methods, see *Netscape Developers JavaScript Reference* at

<http://developer.netscape.com/docs/manuals/communicator/jsref/>

index.html?content=core1.htm#1080770 and *A Primer on JavaScript Arrays*
by Danny Goodman at
http://developer.netscape.com/viewsource/goodman_arrays.html.