JAVA SCRIPT
FOR DUMMIES

by Emily A. Vander Veer

A Quick Reference for the Rest of Us!

The Quick and Easy Way to Remember JavaScript Commands and Features

Step-by-Step Instructions for Creating Interactive Web Pages Quickly and Easily — All New!

Practical HTML and JavaScript Tricks and Tips
# Table of Contents

**Introduction: How to Use This Book** ........................................... 1

- Relating JavaScript to the World Wide Web .................................. 2
- The Internet ...................................................................................... 2
- The World Wide Web .......................................................................... 2
- JavaScript the Magnificent .............................................................. 3
- Finding Exactly What You’re Looking for in This Book .................. 4
- Understanding the Conventions Used in This Book ....................... 5
- Figuring Out What the Icons Mean .................................................. 7
- Where to Go from Here ..................................................................... 8

**Part I: First Things First: Creating HTML**

**Objects to Work within JavaScript** .............................................. 9

- Creating a Meat-and-Potatoes Web Page ........................................ 10
- Body section: `<BODY>` . . . `<BODY>` ........................................... 11
- Header section: `<HEAD>` . . . `<HEAD>` ........................................ 12
- Interactive form: `<FORM>` . . . `<FORM>` ....................................... 12
- JavaScript script: `<SCRIPT>` . . . `<SCRIPT>` ............................... 13
- Web page: `<HTML>` . . . `<HTML>` ................................................ 14

**Inserting Form Elements into Your Web Page** ............................ 15

- ActiveX component: `<OBJECT>` . . . `<OBJECT>` .......................... 15
- Graphic image: `<IMG>` ................................................................. 16
- HTML comment: `<!-- . . . -->` .................................................... 18
- Frame: `<FRAMESET>` . . . `<FRAME>` . . . `<FRAMESET>` ................. 18
- Hypertext anchor: `<A>` . . . `<A>` .................................................. 19
- Hypertext link: `<A>` . . . `<A>` ...................................................... 20

**Inserting Interactive Form Elements: `<INPUT>`** ......................... 21

- Button .............................................................................................. 22
- Check box ....................................................................................... 22
- fileUpload ...................................................................................... 23
- Hidden ............................................................................................ 23
- Password ......................................................................................... 24
- Radio button ................................................................................... 24
- Reset ............................................................................................... 25
- Select .............................................................................................. 25
- Submit ............................................................................................ 27
- Text ............................................................................................... 27
- Textarea .......................................................................................... 28

**Inserting a Java Applet: `<APPLET>` . . . `<APPLET>` .................... 29

**Mapping an Area: `<MAP>` . . . `<AREA>` . . . `<MAP>` .................... 31

**Plugging in a Plug-in: `<EMBED>` . . . `<EMBED>` ......................... 32

**Specifying Useful Web Page Features** ........................................ 33

- CGI program to call on submit: `<FORM>` . . . `<FORM>` .................. 34
- Title for your Web page: `<TITLE>` . . . `<TITLE>` ............................ 34
# Part II: JavaScript Basics

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About JavaScript Security</td>
<td>36</td>
</tr>
<tr>
<td>Basics of the JavaScript Object Model</td>
<td>36</td>
</tr>
<tr>
<td>Built-in JavaScript data types</td>
<td>37</td>
</tr>
<tr>
<td>Hierarchy of JavaScript objects</td>
<td>37</td>
</tr>
<tr>
<td>Utility objects</td>
<td>39</td>
</tr>
<tr>
<td>Basic Punctuation and Syntax</td>
<td>40</td>
</tr>
<tr>
<td>Nested quotes</td>
<td>40</td>
</tr>
<tr>
<td>Pairs</td>
<td>41</td>
</tr>
<tr>
<td>Spelling and capitalization (case)</td>
<td>41</td>
</tr>
<tr>
<td>Top-down</td>
<td>42</td>
</tr>
<tr>
<td>Comments ('<em>...</em>/ and //&lt;/&gt;)</td>
<td>42</td>
</tr>
<tr>
<td>Conditional Expressions: if...else</td>
<td>43</td>
</tr>
<tr>
<td>Loops</td>
<td>43</td>
</tr>
<tr>
<td>break</td>
<td>44</td>
</tr>
<tr>
<td>continue</td>
<td>44</td>
</tr>
<tr>
<td>for</td>
<td>45</td>
</tr>
<tr>
<td>for...in</td>
<td>46</td>
</tr>
<tr>
<td>while</td>
<td>46</td>
</tr>
<tr>
<td>Operators</td>
<td>47</td>
</tr>
<tr>
<td>Assignment operators</td>
<td>47</td>
</tr>
<tr>
<td>Comparison operators</td>
<td>48</td>
</tr>
<tr>
<td>Logical operators</td>
<td>48</td>
</tr>
<tr>
<td>Mathematical operators</td>
<td>49</td>
</tr>
<tr>
<td>Operator precedence</td>
<td>49</td>
</tr>
<tr>
<td>Special Operators</td>
<td>50</td>
</tr>
<tr>
<td>typeof</td>
<td>50</td>
</tr>
<tr>
<td>void</td>
<td>51</td>
</tr>
<tr>
<td>String operators</td>
<td>52</td>
</tr>
<tr>
<td>Unary operators</td>
<td>52</td>
</tr>
<tr>
<td>Variables</td>
<td>53</td>
</tr>
<tr>
<td>Accessing variables</td>
<td>53</td>
</tr>
<tr>
<td>Assigning values to a variable</td>
<td>53</td>
</tr>
<tr>
<td>Declaring variables</td>
<td>53</td>
</tr>
<tr>
<td>Understanding variable scope</td>
<td>54</td>
</tr>
</tbody>
</table>

# Part III: HTML Objects: The Heart of It All

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About Objects</td>
<td>56</td>
</tr>
<tr>
<td>anchor</td>
<td>56</td>
</tr>
<tr>
<td>applet</td>
<td>57</td>
</tr>
<tr>
<td>area</td>
<td>57</td>
</tr>
<tr>
<td>button</td>
<td>58</td>
</tr>
<tr>
<td>checkbox</td>
<td>59</td>
</tr>
<tr>
<td>document</td>
<td>59</td>
</tr>
<tr>
<td>fileUpload</td>
<td>60</td>
</tr>
<tr>
<td>form</td>
<td>61</td>
</tr>
<tr>
<td>frame</td>
<td>61</td>
</tr>
<tr>
<td>hidden</td>
<td>62</td>
</tr>
<tr>
<td>history</td>
<td>62</td>
</tr>
<tr>
<td>Table of Contents</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>image ........................................ 63</td>
<td></td>
</tr>
<tr>
<td>link ........................................... 64</td>
<td></td>
</tr>
<tr>
<td>location ................................. 64</td>
<td></td>
</tr>
<tr>
<td>Math ........................................ 65</td>
<td></td>
</tr>
<tr>
<td>navigator .................................... 66</td>
<td></td>
</tr>
<tr>
<td>password ................................... 66</td>
<td></td>
</tr>
<tr>
<td>plugin ...................................... 67</td>
<td></td>
</tr>
<tr>
<td>radio ........................................ 67</td>
<td></td>
</tr>
<tr>
<td>reset ......................................... 68</td>
<td></td>
</tr>
<tr>
<td>select ....................................... 68</td>
<td></td>
</tr>
<tr>
<td>Submit ....................................... 69</td>
<td></td>
</tr>
<tr>
<td>text .......................................... 69</td>
<td></td>
</tr>
<tr>
<td>textarea .................................... 70</td>
<td></td>
</tr>
<tr>
<td>window ....................................... 70</td>
<td></td>
</tr>
</tbody>
</table>

**Part IV: Data Types: Building Basic**

**JavaScript Objects** ........................................ 73

| Array ........................................ 74 |
| Date ......................................... 75 |
| Function .................................... 77 |
| Option ....................................... 77 |
| String ....................................... 79 |

**Part V: Functions** ........................................ 81

| About Using Functions ..................... 82 |
| Calling a function .......................... 82 |
| Defining a function ......................... 83 |
| Returning values ............................. 83 |

| Built-In Functions ......................... 84 |
| escape() — encodes a string ............... 84 |
| eval() — evaluates a string ............... 84 |
| isNaN() — is not a number .................. 85 |
| parseFloat() — turns strings into floating-point numbers ............... 85 |
| parseInt() — turns strings into integers .... 86 |
| unescape() — decodes a string ............. 87 |

| Creating a Custom JavaScript Function .... 88 |
| With a fixed number of parameters ........ 88 |
| With a variable number of parameters .... 88 |

| Functions that Help You Create and Work with Objects .......... 89 |
| new .......................................... 89 |
| this ......................................... 90 |
| with ......................................... 90 |

**Part VI: Methods: How an Object Behaves** .......... 91

<p>| Methods ...................................... 92 |
| abs .......................................... 92 |
| acos ......................................... 93 |
| alert ........................................ 93 |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor</td>
<td>93</td>
</tr>
<tr>
<td>asin</td>
<td>94</td>
</tr>
<tr>
<td>atan</td>
<td>94</td>
</tr>
<tr>
<td>atan2</td>
<td>94</td>
</tr>
<tr>
<td>back</td>
<td>94</td>
</tr>
<tr>
<td>big</td>
<td>95</td>
</tr>
<tr>
<td>blink</td>
<td>95</td>
</tr>
<tr>
<td>blur</td>
<td>95</td>
</tr>
<tr>
<td>bold</td>
<td>96</td>
</tr>
<tr>
<td>ceil</td>
<td>96</td>
</tr>
<tr>
<td>charAt</td>
<td>97</td>
</tr>
<tr>
<td>clearTimeout</td>
<td>97</td>
</tr>
<tr>
<td>click</td>
<td>97</td>
</tr>
<tr>
<td>close (document)</td>
<td>98</td>
</tr>
<tr>
<td>close (window)</td>
<td>98</td>
</tr>
<tr>
<td>confirm</td>
<td>98</td>
</tr>
<tr>
<td>cos</td>
<td>99</td>
</tr>
<tr>
<td>exp</td>
<td>99</td>
</tr>
<tr>
<td>fixed</td>
<td>99</td>
</tr>
<tr>
<td>floor</td>
<td>100</td>
</tr>
<tr>
<td>focus</td>
<td>100</td>
</tr>
<tr>
<td>fontColor</td>
<td>101</td>
</tr>
<tr>
<td>fontSize</td>
<td>101</td>
</tr>
<tr>
<td>forward</td>
<td>101</td>
</tr>
<tr>
<td>getDate</td>
<td>102</td>
</tr>
<tr>
<td>getDay</td>
<td>102</td>
</tr>
<tr>
<td>getHours</td>
<td>102</td>
</tr>
<tr>
<td>getMinutes</td>
<td>103</td>
</tr>
<tr>
<td>getMonth</td>
<td>103</td>
</tr>
<tr>
<td>getSeconds</td>
<td>103</td>
</tr>
<tr>
<td>getTime</td>
<td>104</td>
</tr>
<tr>
<td>getTimezoneOffset</td>
<td>104</td>
</tr>
<tr>
<td>getYear</td>
<td>104</td>
</tr>
<tr>
<td>go</td>
<td>105</td>
</tr>
<tr>
<td>indexOf</td>
<td>105</td>
</tr>
<tr>
<td>italics</td>
<td>106</td>
</tr>
<tr>
<td>javaEnabled()</td>
<td>106</td>
</tr>
<tr>
<td>join</td>
<td>106</td>
</tr>
<tr>
<td>lastIndexof</td>
<td>107</td>
</tr>
<tr>
<td>link</td>
<td>107</td>
</tr>
<tr>
<td>log</td>
<td>108</td>
</tr>
<tr>
<td>max</td>
<td>108</td>
</tr>
<tr>
<td>min</td>
<td>108</td>
</tr>
<tr>
<td>open (document)</td>
<td>108</td>
</tr>
<tr>
<td>open (window)</td>
<td>109</td>
</tr>
<tr>
<td>parse</td>
<td>110</td>
</tr>
<tr>
<td>pow</td>
<td>110</td>
</tr>
<tr>
<td>prompt</td>
<td>110</td>
</tr>
<tr>
<td>random</td>
<td>111</td>
</tr>
<tr>
<td>reload</td>
<td>111</td>
</tr>
<tr>
<td>replace</td>
<td>111</td>
</tr>
</tbody>
</table>
reset ................................................................. 112
reverse .............................................................. 112
round ................................................................. 112
scroll ................................................................. 113
select ................................................................. 113
setDate .............................................................. 113
setHours ............................................................. 114
setMinutes .......................................................... 114
setMonth ............................................................. 114
setSeconds .......................................................... 115
setTime ............................................................... 115
setTimeout ........................................................... 115
setYear ............................................................... 116
sin ...................................................................... 116
small ................................................................. 117
sort ................................................................. 117
split ................................................................. 118
sqrt ................................................................. 118
strike ............................................................... 118
sub ................................................................. 119
submit .............................................................. 119
substring ........................................................... 119
sup ................................................................. 120
tan ................................................................. 120
toGMTString ........................................................ 120
toLocaleString ..................................................... 121
toLowerCase ......................................................... 121
toString ............................................................ 121
toUpperCase ......................................................... 122
UTC ................................................................. 122
write ............................................................... 122
writeln .............................................................. 123

Part VII: Properties (Object Data) .................. 125

About Properties .................................................. 126
action ............................................................... 126
alinkColor .......................................................... 126
appCodeName ....................................................... 127
applets ............................................................. 127
appName ............................................................ 127
appVersion ........................................................ 127
arguments .......................................................... 127
bgColor ............................................................. 128
border ............................................................. 128
checked (checkbox) .............................................. 128
checked (radio) .................................................... 128
complete .......................................................... 128
cookie ............................................................. 129
defaultChecked (checkbox) .................................. 129
defaultChecked (radio) ........................................ 129
<table>
<thead>
<tr>
<th>Variable</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>defaultSelected</code></td>
<td>129</td>
</tr>
<tr>
<td><code>defaultStatus</code></td>
<td>129</td>
</tr>
<tr>
<td><code>defaultValue (password)</code></td>
<td>130</td>
</tr>
<tr>
<td><code>defaultValue (text)</code></td>
<td>130</td>
</tr>
<tr>
<td><code>defaultValue (textarea)</code></td>
<td>130</td>
</tr>
<tr>
<td><code>description</code></td>
<td>130</td>
</tr>
<tr>
<td><code>E</code></td>
<td>130</td>
</tr>
<tr>
<td><code>elements</code></td>
<td>130</td>
</tr>
<tr>
<td><code>embeds</code></td>
<td>131</td>
</tr>
<tr>
<td><code>enabledPlugin</code></td>
<td>131</td>
</tr>
<tr>
<td><code>encoding</code></td>
<td>131</td>
</tr>
<tr>
<td><code>fgColor</code></td>
<td>131</td>
</tr>
<tr>
<td><code>filename</code></td>
<td>132</td>
</tr>
<tr>
<td><code>forms</code></td>
<td>132</td>
</tr>
<tr>
<td><code>frames</code></td>
<td>132</td>
</tr>
<tr>
<td><code>hash</code></td>
<td>132</td>
</tr>
<tr>
<td><code>height</code></td>
<td>133</td>
</tr>
<tr>
<td><code>host</code></td>
<td>133</td>
</tr>
<tr>
<td><code>hostname</code></td>
<td>133</td>
</tr>
<tr>
<td><code>href</code></td>
<td>133</td>
</tr>
<tr>
<td><code>hspace</code></td>
<td>134</td>
</tr>
<tr>
<td><code>images</code></td>
<td>134</td>
</tr>
<tr>
<td><code>index</code></td>
<td>134</td>
</tr>
<tr>
<td><code>lastModified</code></td>
<td>134</td>
</tr>
<tr>
<td><code>length</code></td>
<td>135</td>
</tr>
<tr>
<td><code>linkColor</code></td>
<td>136</td>
</tr>
<tr>
<td><code>links</code></td>
<td>136</td>
</tr>
<tr>
<td><code>LN2</code></td>
<td>136</td>
</tr>
<tr>
<td><code>LN10</code></td>
<td>136</td>
</tr>
<tr>
<td><code>location</code></td>
<td>137</td>
</tr>
<tr>
<td><code>LOG2E</code></td>
<td>137</td>
</tr>
<tr>
<td><code>LOG10E</code></td>
<td>137</td>
</tr>
<tr>
<td><code>lowsrc</code></td>
<td>137</td>
</tr>
<tr>
<td><code>method</code></td>
<td>137</td>
</tr>
<tr>
<td><code>mimeTypes</code></td>
<td>138</td>
</tr>
<tr>
<td><code>name</code></td>
<td>138</td>
</tr>
<tr>
<td><code>opener</code></td>
<td>139</td>
</tr>
<tr>
<td><code>options</code></td>
<td>139</td>
</tr>
<tr>
<td><code>parent (frame)</code></td>
<td>139</td>
</tr>
<tr>
<td><code>parent (window)</code></td>
<td>139</td>
</tr>
<tr>
<td><code>pathname</code></td>
<td>140</td>
</tr>
<tr>
<td><code>PI</code></td>
<td>140</td>
</tr>
<tr>
<td><code>plugins</code></td>
<td>140</td>
</tr>
<tr>
<td><code>port</code></td>
<td>140</td>
</tr>
<tr>
<td><code>protocol</code></td>
<td>141</td>
</tr>
<tr>
<td><code>prototype</code></td>
<td>141</td>
</tr>
<tr>
<td><code>referrer</code></td>
<td>142</td>
</tr>
<tr>
<td><code>search (link)</code></td>
<td>142</td>
</tr>
<tr>
<td><code>search (location)</code></td>
<td>142</td>
</tr>
<tr>
<td><code>selected</code></td>
<td>143</td>
</tr>
<tr>
<td><code>selectedIndex (options)</code></td>
<td>143</td>
</tr>
</tbody>
</table>
How to Use This Book

You loved the book (JavaScript For Dummies, that is), and you saw the movie six times. You want to use JavaScript to add flash and interactivity to your Web pages and, more than likely, you have a pretty good idea of how you want to go about it. Still, you could use a handy reference manual to keep nearby for those times when you know what you want to do, but can't remember exactly how to do it. Well, you're in luck! Because you're obviously reading this, congratulate yourself: You've found the most complete, down-to-earth JavaScript reference available in this or any other galaxy.

Unlike some reference works that are technically correct but hopelessly detailed, this book offers tons of working examples of how to do the things you really want to do with JavaScript, like adding multimedia and interactive input forms to your Web pages.
How to Use This Book

JavaScript For Dummies Quick Reference gives you a bare-bones HTML primer that explains the subset of HTML statements that are absolutely indispensable to a JavaScripter, followed by a comprehensive review of the JavaScript language itself and an in-depth look at the JavaScript object model — every last object, property, method, event handler, and function available in JavaScript.

Relating JavaScript to the World Wide Web

The World Wide Web (from now on out, called just plain Web) is an extremely complex animal. There’s so much to know — from communication protocols to browsers, from HTML to JavaScript to Java, and other proprietary architectures and platforms. Unfortunately, most of these technologies are described and documented as though they exist in a vacuum, which leaves the average intelligent person curious about the big picture: How does it all fit together? If that’s a tree, what does the forest look like? In short, what’s it all about, Alifie?

You really don’t have to be a rocket scientist to understand this JavaScript stuff — all you need is an overview in plain English. Here, then, is a birds-eye view of the role JavaScript plays in this brave new world of Web application development.

The Internet

In the beginning, there was the Internet: A huge (and I mean huge) conglomeration of networked computers. At first, you did have to be a rocket scientist (or at least a computer scientist) to access the Internet, because its interface was so cryptic and unfriendly to humans. That all changed with the advent of the Internet protocols that formed the World Wide Web.

The World Wide Web

The Web runs primarily on two protocols:

- HTML (HyperText Markup Language): The language in which Web pages are written
- HTTP (HyperText Transfer Protocol): The special way of communicating that lets computers like Web servers and Web clients shuttle Web pages back and forth

A Web server is an industrial-strength computer, usually running UNIX or Windows NT. Web servers hold peoples’ Web pages and hang around waiting for requests. A Web client is a regular computer

(usually a PC or a Macintosh) that has Web browser software installed on it — browsers like Netscape Navigator or Microsoft Internet Explorer that hunt and peck through cyberspace at your command.

When you use your favorite Web browser to load a Web page, such as the one denoted by the URL (Uniform Resource Locator) http://www.dummies.com, your browser fetches the page from wherever it is on the Web and brings it to your browser. Your browser then interprets the fetched HTML file and displays it for you in living color.

JavaScript the Magnificent

Where does JavaScript come in? JavaScript is a scripting language designed by Netscape Communications. A scripting language is a limited programming language designed to extend the capabilities of another application. The JavaScript language is an extension to HTML and lets you create interactive Web pages. With JavaScript-enabled Web pages, users can press buttons, type text, perform calculations, and call Java programs or plug-ins — without a whole lot of programming effort. (Other ways exist to create interactive Web pages, notably by using server-side scripting by means of CGI, or Common Gateway Interface programs. But I’m here to tell you, you’re back to needing a PhD in astrophysics when you start dabbling around with CGI.)

At the time of this writing, JavaScript is supported by Netscape Navigator 3.0 and Internet Explorer 3.0. (I say that these browsers support JavaScript because they have JavaScript interpreters built into them.) Yet, while Netscape Navigator and Internet Explorer both support JavaScript, their implementations differ slightly. I alert you throughout the book when I cover topics that the two browsers handle differently. To be on the safe side, you should always test your JavaScript-enabled Web pages on both Web browsers before you send them out into the big wide world.

The JavaScript implementations on various Web browsers have been known to change slightly from version to version (for example, from Netscape Navigator 2.0 to 3.0) — so you can probably expect this situation to continue in the future. What this fact means to you as a JavaScript author is that it’s a good idea to test your JavaScript-enabled Web pages on the new Web browser versions as soon as they become available. You want to make sure that the Web pages that you created for the older Web browser still work on the newer version. After all, if users attempt to load your Web page and it tanks, you’re the one they’ll blame!
Finding Exactly What You're Looking for in This Book

This book is organized so that you can find information quickly, either by browsing through the main sections (described in the following paragraphs) or by looking through the alphabetized Table of Contents.

Part I, "First Things First: Creating HTML Objects to Work within JavaScript," presents the basic HTML statements that you need to create a basic Web page from scratch. Many of the statements create the actual HTML elements, or objects, which you can change, look at, display, and otherwise manipulate with JavaScript statements.

Part II, "JavaScript Basics," presents the JavaScript object model and shows you how to reference each object in the object hierarchy. This part also introduces the JavaScript keywords, explains how you can use each keyword, and describes the syntax you need to follow to create valid JavaScript statements.

Part III, "HTML Objects: The Heart of It All," gives you a blow-by-blow analysis of all of the HTML objects available to you in JavaScript (in other words, the ones you learn how to create in Part I). This part, together with parts IV, V, VI, VII, and VIII, provide you a complete JavaScript language reference.

Part IV, "Data Types: Building Basic JavaScript Objects," shows you how to put the most basic JavaScript objects into your Web pages. This part covers the JavaScript objects that you're likely to include in almost every script you're likely to write — objects like Array, Date, Function, Option, and String. This part shows you the correct syntax for including basic objects in your scripts and even gives you examples of working JavaScript code.

Part V, "Functions," lists all of JavaScript's built-in functions, describes how to use functions to create instances of objects, and clues you in on how to customize your own functions.

Part VI, "Methods: How an Object Behaves," shows you all you'd ever want to know about methods. This part gives you a rundown of every method available to you in the JavaScript library, demonstrates the proper way to call each method, and gives you easy-to-understand code examples and explanations.

Part VII, "Properties (Object Data)," is an exhaustive listing of the items that represent an object's data — its properties. This part tells you how to access and modify object properties and relates those properties to their corresponding HTML tag attributes.

Part VIII, "Event Handlers," shows you how to use these special attributes to specify JavaScript statements (typically you'll specify just one statement — a function call) for the browser to invoke automatically whenever an event occurs.

Part IX, "Cool Things You Can Do with JavaScript," shows you how to do all the stuff you really want to do with JavaScript — things like adding multimedia action to your Web pages, creating interactive controls, validating user input, and calling other (non-JavaScript) components from your JavaScript scripts. In this part, you find instructions for everything from loading a JavaScript-enabled Web page (which is dead-easy) to interacting with cookies (a considerably more advanced topic).

Appendix A, "Reserved Words," briefs you on the JavaScript keywords that you need to avoid when you name your variables and functions.

Appendix B, "Color Values," contains a list of predefined shades that you can use to color the background (or any other part) of your Web page.

The glossary, called "Techie Talk," appears at the very end of the book. You can check this part to look up words you're not familiar with, or that you've come across before but have forgotten.

Understanding the Conventions Used in This Book

HTML and JavaScript code appear in this book in monospaced font, like this line of HTML code:

<TITLE>My First JavaScript-Enabled Web Page</TITLE>

Make sure that you follow word-for-word the syntax in the examples (that is, the spacing, capitalization, and spelling). Some variations may work, but some won't — and consistency pays big dividends in reading your code later. The exception to this syntax rule relates to italicized words, which act as placeholders for other values, which you can substitute in your actual code. For example, in the following line of JavaScript code, you can replace myCatName and "Scooter Pie" with other values, but you need to type var exactly the way it appears:

```
var myCatName = "Scooter Pie"
```
If you need to type something in, the directive appears in boldface, like this:

Type return.

I also use boldface to highlight lines of code I refer to in the text.

Due to the narrow margins of this book, sometimes code examples may wrap around from one line to another. JavaScript doesn’t require a line continuation character, so you can copy the code exactly as it appears, unless the break occurs between two quotes, like this:

```
var favoriteActivity = "sleeping late on Sundays" // Warning! Invalid break
```

And finally, because JavaScript scripts don’t exist in a vacuum (you must integrate your JavaScript scripts into HTML files), you see two different types of coding statements in this book:

- HTML statements
- JavaScript statements

The conventions listed in the preceding paragraphs are all you need to follow to create customized, working JavaScript statements on your own computer. HTML statements, however, tend to be a little more daunting until you’re familiar with them. The following code shows the correct syntax for an HTML statement, and the following table explains the HTML elements in more detail. Here’s how you make sense of HTML syntax:

```
<TAG-NAME ATTRIBUTE[="VALUE"]...[some text] ["</TAG-NAME>
```

<table>
<thead>
<tr>
<th>HTML Syntax Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Angle brackets (<code>&lt;&gt;</code>) surround a tag name.</td>
</tr>
<tr>
<td>TAG-NAME</td>
<td>Tag names are HTML keywords like TITLE, SCRIPT, and INPUT.</td>
</tr>
<tr>
<td><code>[</code></td>
<td>Square brackets surround items that are optional (depending on the tag name that you’re working with).</td>
</tr>
<tr>
<td>ATTRIBUTE</td>
<td>Attributes are associated with specific tag names. For example, the TYPE attribute is associated with the INPUT tag.</td>
</tr>
<tr>
<td><code>=&quot;VALUE&quot;</code></td>
<td>Some attributes require values. For example, the TYPE attribute of the INPUT tag requires a value which could be BUTTON, RESET, SUBMIT, or some other predefined value.</td>
</tr>
</tbody>
</table>

---

**Figuring Out What the Icons Mean**

To make your life easier (as it relates to JavaScript, at least!), I include a few icons to give you a heads-up on practical scripting-related tidbits:

This icon indicates a handy cross-reference to some other For Dummies book that explains the related topic in lots more detail.

This icon alerts you to handy tricks and techniques that can save you time, hassle, and many, many tufts of hair.

Next to this icon are common pitfalls, bugs, and assorted “uh-ohs” to look out for.

This icon points out items that don’t work the way you’d expect them to — a touch of illogic in the normally logical world of computerdom.

This icon marks a faster way that you might choose to accomplish a task — saving you valuable time and keystrokes.

This icon flags Web sites that you can visit for more in-depth information on a related topic.

Rejoice when you see this icon! It indicates real-live, working JavaScript/HTML source code.
Where to Go from Here

Because JavaScript gives you instant feedback, it’s really fun to use (okay, it’s fun compared to other programming languages — maybe not compared to a two-week beach vacation!). To get the most out of this book, you may want to try creating a Web page from scratch and then adding to it a piece at a time as ideas occur to you. Take the afternoon off, kick the dogs out, and experiment with some of the suggestions in this book. If you get stuck, just flip to the section that covers whatever’s giving you a hard time, copy the example you’ll find there, and modify it until it’s just the way you want it. Creating incredible Web pages with JavaScript isn’t illegal, immoral, or fattening, so what are you waiting for? Go for it!

First Things First: Creating HTML Objects to Work within JavaScript

JavaScript is an ultra- pared-down, ultra-easy programming language specifically designed to make Web page elements interactive. An interactive element is one that responds to user input — for example, a push button that causes something to happen when a user clicks on it, or a text field that automatically checks the accuracy of text that a user types in. In order to make elements interactive, though, the elements first have to exist. For elements to exist, you have to create them — which you can do easily with HTML statements like the ones in this part.

In this part . . .

» Creating a basic, no-nonsense Web page in HTML
» Adding elements to your Web page
» Connecting your Web page to a server-side CGI program
Creating a Meat-and-Potatoes Web Page

Creating a Web page is a fairly simple process, as you can see in this overview. (The rest of this part shows you exactly how to implement each phase of the overview.)

First, you need to create an HTML document file called somefile.html (or, if your system doesn’t support long filename extensions, name your file somefile.htm). You can use whichever text editor you’re most comfortable with to create the HTML file, such as WordPad, BBEdit, the text editor that’s included as part of Netscape Navigator Gold, or even your favorite word processor package — as long as the program lets you save plain text. The HTML document that you create should have the following sections:

- **The overall document section (required):** In order for the HTML interpreter to recognize that your document is an HTML file and not just any old file, the document file’s first line must contain the beginning `<HTML>` tag, and the last line must contain the ending `</HTML>` tag. All the other HTML tags and JavaScript statements you decide to incorporate into your Web page are optional — but whichever tags or statements you include must appear between the `<HTML>`...`</HTML>` tags.

- **The header section (optional):** If your document has a header (such as “Creating a Meat-and-Potatoes Web Page,” which you see at the beginning of this section), the header immediately follows the beginning `<HTML>` tag. The header is bounded by the opening `<HEAD>` tag and the closing `</HEAD>` tag. You must place some elements inside the header, such as your Web page’s title. Other elements — for example, a JavaScript script, which is identified by the `<SCRIPT>` tag — may or may not take up residence in the header.

- **The body section (technically optional but practically required):** You could create a document that doesn’t contain a body, but it would be pretty boring! All the interactive form elements that you become familiar with in this part must appear between the `<BODY>`...`</BODY>` tags in order to be recognized by the HTML interpreter.

See also “Getting Started with a Bare-Bones HTML Template,” in Part IX, which contains a mini-HTML file with the document, header, body, and script tags already in place.

After you create an HTML document file, you can load the page in any JavaScript-enabled Web browser (for example, Netscape Navigator or Microsoft Internet Explorer) and view your new Web page.

When you finish testing your Web page and it looks absolutely perfect, you’ll probably want to share it with the world — which requires giving your document file (and perhaps some cash) to a Web server administrator. *Creating Web Pages For Dummies*, by Bud Smith and Arthur Bek (published by IDG Books Worldwide, Inc.), contains a whole chapter describing free (!) and easy ways to get your page up and running on the Web.

**Body section: `<BODY>`...`</BODY>`**

Typically, the body of the document is where most of the HTML statements live — smack-dab between the `<BODY>`...`</BODY>` tag pairs.

To define a body section in HTML, use this tag pair:

```html
<BODY>...
</BODY>
```

To access that body section in JavaScript, use this identifier:

```javascript
document

The `<BODY>`...`</BODY>` tag pair, itself, gives you the opportunity to customize the colors that your Web page displays.

You can change the color of any of the items that appear in the Description column of the following table to any of the color values shown in Appendix B.

<table>
<thead>
<tr>
<th>Setting Colors in HTML</th>
<th>Description</th>
<th>Accessing Colors in JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGCOLOR=&quot;chartreuse&quot;</td>
<td>Background page color</td>
<td>document.bgColor</td>
</tr>
<tr>
<td>TEXT=&quot;blue&quot;</td>
<td>Foreground (text) color</td>
<td>document.fgColor</td>
</tr>
<tr>
<td>LINK=&quot;yellow&quot;</td>
<td>Link color before it's clicked</td>
<td>document.linkColor</td>
</tr>
<tr>
<td>ALINK=&quot;white&quot;</td>
<td>Link color as it's being clicked</td>
<td>document.alinkColor</td>
</tr>
<tr>
<td>VLINK=&quot;black&quot;</td>
<td>Link color after it's been clicked</td>
<td>document.vlinkColor</td>
</tr>
</tbody>
</table>

Here’s how the code in the preceding table might look when you set up the body of an HTML document:

```html
<html>

<body

BGColor="someColor"
TEXT="someColor"
LINK="someColor"
ALINK="someColor"
VLINK="someColor">

...```
Most of your HTML code, including code for your interactive form, goes right here.

</BODY>
</HTML>

Header section: <HEAD> . . . </HEAD>

Because the JavaScript interpreter (the software bundled with some Web browsers that enables the browser to recognize JavaScript statements) begins reading the HTML code at the top of each file and works downward, the <HEAD> tag should appear near the top of the file:

<HTML>
<HEAD>
Statements placed here are interpreted as soon as the Web page is loaded.
</HEAD>

The bulk of your HTML statements, including your body section, goes here.

</HTML>

Take advantage of the organizational convenience of the header and body sections of an HTML document. For example, if you make a habit of putting all your JavaScript function definitions in the header section, you can be sure that your functions will be available to the rest of your Web page.

Interactive form: <FORM> . . . </FORM>

The <FORM> . . . </FORM> tag pair is arguably one of the most important tags you can use. JavaScript maven fantastique. The <FORM> tag surrounds the heart of the JavaScript-enabled Web page: the interactive form. A form is used to gather input from users and to send, or post, form data to a server for additional processing. You always define forms inside the </BODY> . . . </BODY> section.

To define a form in HTML, use this tag pair:

<FORM NAME="myForm"> . . . </FORM>

To access that form in JavaScript, use this identifier:

document.myForm

You must place the tag for any form element you choose to include in your Web page, be it a button, a list box, a text field, or whatever, between the <FORM> . . . </FORM> tags, as shown in the following table:

<table>
<thead>
<tr>
<th>Tag Syntax for &lt;FORM&gt;</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;FORM&gt;</td>
<td>Opening &lt;FORM&gt; tag</td>
</tr>
<tr>
<td>NAME=&quot;formName&quot;</td>
<td>Internal name of form (for coding purposes)</td>
</tr>
<tr>
<td>[TARGET=&quot;windowName&quot;]</td>
<td>Window to display server response (optional; default is current window)</td>
</tr>
<tr>
<td>[ACTION=&quot;serverURL&quot;]</td>
<td>URL where form data is sent when the user submits the form (optional)</td>
</tr>
<tr>
<td>[METHOD=GET</td>
<td>POST]</td>
</tr>
<tr>
<td>[ENCODING=&quot;encodingType&quot;]</td>
<td>Special data encoding scheme, if any (optional)</td>
</tr>
<tr>
<td>[onSubmit=&quot;handlerText&quot;]</td>
<td>Code to invoke when form is submitted (optional)</td>
</tr>
<tr>
<td>. . .</td>
<td>Put all your form elements here</td>
</tr>
<tr>
<td>&lt;/FORM&gt;</td>
<td>Closing &lt;FORM&gt; tag</td>
</tr>
</tbody>
</table>

See also "Inserting Form Elements into Your Web Page," later in this part, which defines form elements like buttons, text fields, and list boxes.

You may never take advantage of all the stuff in the syntax; at least, not until you become an expert! To get you started, here's a common, everyday example of the syntax for an interactive form:

<FORM NAME="myForm">

<input TYPE="button" NAME="calculatePrice"
VALUE="Calculate Now!"
onClick="calculatePrice()">

</FORM>

The syntax for HTML in general, and the <INPUT> tag in particular, can be a little intimidating if you're not familiar with it. For the whole scoop on HTML syntax, check out HTML For Dummies, 2nd Edition, by Ed Tittel and Steve James.

JavaScript script: <SCRIPT> . . . </SCRIPT>

The <SCRIPT> . . . </SCRIPT> HTML tag pair lets you insert a JavaScript script into your Web page. (Be aware that JavaScript isn't the only scripting language that you can specify; that's why the <SCRIPT> tag has an associated LANGUAGE attribute.) You can place a JavaScript script inside the header or the body section of your Web page, and you can insert as many scripts as you want. (See also Part IX for some cool script ideas.)
14 Creating a Meat-and-Potatoes Web Page

Because JavaScript is an object-based (not object-oriented) language, the JavaScript code you write will be rife with references to entities such as objects, methods, and properties. Unfortunately, due to space constraints, this book doesn’t contain a discussion of object-based programming. If you’re interested in object-based programming, concepts, and techniques (especially with respect to JavaScript), pick up a copy of JavaScript For Dummies, by Emily Vander Veer (that’s me, by the way).

<table>
<thead>
<tr>
<th>Tag Syntax for &lt;SCRIPT&gt;</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;SCRIPT&gt;</td>
<td>Opening &lt;SCRIPT&gt; tag</td>
</tr>
<tr>
<td>[LANGUAGE=&quot;JavaScript&quot;]</td>
<td>Scripting language (optional but recommended; JavaScript is the default)</td>
</tr>
<tr>
<td>[SRC=&quot;fileName.js&quot;]&gt;</td>
<td>JavaScript source filename (optional)</td>
</tr>
<tr>
<td>...</td>
<td>Your JavaScript statements go here unless you specify a value for SRC</td>
</tr>
<tr>
<td>&lt;/SCRIPT&gt;</td>
<td>Closing &lt;SCRIPT&gt; tag</td>
</tr>
</tbody>
</table>

Here’s an example of the <SCRIPT> tag in action, just to get you started:

```html
<HTML>
  <SCRIPT LANGUAGE="JavaScript">
    <![CDATA[
      function calculateTotal(
        cost, 
        taxRate)
      {
        // Use the newly defined function to calculate the total cost. 
        total = cost + (cost * taxRate); 
        // Return the total cost. 
        return total;
      }
    ]]>
  </SCRIPT>
</HTML>
```

Make sure that you don’t place HTML statements inside the <SCRIPT>...</SCRIPT> tags. The only statements that the JavaScript interpreter considers valid between these two tags are JavaScript statements — any other type of statement causes a syntax error.

**Web page: <HTML> ... </HTML>**

Some browsers may load your HTML document file just to find out if either or both of these tags are missing. Technically, though, the Web browsers don’t have to — and there’s no guarantee that they’ll keep doing so in the future. So, to be on the safe side, always include the <HTML>...</HTML> tag pair in your HTML document file. The beginning tag should be the first line in your file, and the ending tag should be the last line, as shown in the following example:

```html
<HTML>
Place any HTML or JavaScript statements that you choose to include in your HTML file between these tags.
</HTML>
```

15 Inserting Form Elements into Your Web Page

Inserting Form Elements into Your Web Page

All the elements described in this section are form elements and, because they’re part of a form, they must be defined between the <FORM>...</FORM> tags, which, in turn, must be defined between the <BODY>...</BODY> tags. Read on for specific examples.

**ActiveX component: <OBJECT> ... </OBJECT>**

ActiveX components are currently compatible only with Internet Explorer’s version of JavaScript, but rumor has it that ActiveX compatibility is coming soon to Navigator.

To embed an ActiveX component in an HTML document, use this tag pair: <OBJECT> ... </OBJECT>

To access that ActiveX component in the Internet Explorer implementation of JavaScript, use this identifier:

document.myForm.myActiveXId

The ink on the <OBJECT> tag specification isn’t even close to being dry at the time of this writing, which means that the information in this section may have changed slightly by the time you read this. To keep up with the official <OBJECT> tag goings-on, keep an eye on the following URL:

http://www.w3.org/pub/WWW/TR/WD-object.html#object

<table>
<thead>
<tr>
<th>Tag Syntax for Embedded ActiveX Component</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;OBJECT&gt;</td>
<td>Opening &lt;OBJECT&gt; tag</td>
</tr>
<tr>
<td>CLASSID=&quot;classid&quot;</td>
<td>Classid of the component</td>
</tr>
<tr>
<td>ID=&quot;componentId&quot;</td>
<td>Internal name of component for coding purposes</td>
</tr>
<tr>
<td>[HEIGHT=&quot;pixels&quot;</td>
<td>&quot;value&quot;]</td>
</tr>
<tr>
<td>[WIDTH=&quot;pixels&quot;</td>
<td>&quot;value&quot;]</td>
</tr>
<tr>
<td>[HSPACE=&quot;pixels&quot;]</td>
<td>Horizontal space between left and right sides of the component and left and right edges of the window, in pixels</td>
</tr>
</tbody>
</table>

(continued)
Inserting Form Elements into Your Web Page

Tag Syntax for Embedded ActiveX Component

[VSPACE="pixels"]
Vertical space between top and bottom of component and top and bottom edges of the window, in pixels

[ALIGN="position"]>
Specifies alignment of image

You can repeat the following optional section as many times as necessary:

[<PARAM>
NAME="parameterName"
VALUE="parameterValue">
Closing <PARAM> tag

</OBJECT>
Closing </OBJECT> tag

The following code embeds an ActiveX component whose purpose in life is to load a Web page inside a Web page (in this example, the IDG Books Worldwide home page). This code then flips the Web page around and around inside a skinny little horizontal window.

<OBJECT
ALIGN=CENTER
CLASSID="clsid:1a4da620-6217-11cf-be62-0080c72eed2d"
WIDTH=650 HEIGHT=40 BORDER=1 HSPACE=5
ID=marquee>
<PARAM NAME="ScrollStyleX" VALUE="Circular">
<PARAM NAME="ScrollStyleY" VALUE="Circular">
<PARAM NAME="szURL" VALUE="http://www.idgbooks.com">
<PARAM NAME="ScrollDelay" VALUE=100>
<PARAM NAME="LoopsX" VALUE=->
<PARAM NAME="LoopsY" VALUE=->
<PARAM NAME="ScrollPixelsX" VALUE=0>
<PARAM NAME="ScrollPixelsY" VALUE=30>
<PARAM NAME="DrawImmediately" VALUE=1>
<PARAM NAME="Whitespace" VALUE=0>
<PARAM NAME="PageFlippingOn" VALUE=1>
<PARAM NAME="Zoom" VALUE=100>
<PARAM NAME="WidthOfPage" VALUE=640>
</OBJECT>

Graphic image: <IMG>

To embed an image in HTML, use this tag:

<IMG NAME="myImage">...

To access that image in Navigator's implementation of JavaScript, use this identifier: document.myForm.myImage.

You can embed any image file into a Web page by using the <IMG> tag. As you may expect, the position of the embedded image is determined by the position of the <IMG> statement in the HTML file. For example, if you define a push button, an embedded image, and a set of radio buttons, in that order, that's the order in which they appear on your Web page.

Tag Syntax for <IMG>

<IMG
NAME="imageName"
SRC="location"
[LOWSRC="location"]
[HEIGHT="pixels"] | ["value"]%
[WIDTH="pixels"] | ["value"]%
[HSPACE="pixels"]
[VSPACE="pixels"]

Opening <IMG> tag
Name of image for internal coding purposes
URL of image to load
URL of alternative low-resolution version of the image
Height of image, in pixels or as a percentage of window height
Width of image, in pixels or as a percentage of window height
Horizontal space between left and right sides of the image, and the left and right edges of the window, in pixels
Vertical space between top and bottom of image and top and bottom edges of window, in pixels
Width of the image's border, if any, in pixels
Specifies alignment of image
Whether the image is a server-side map
Whether the image is a client-side map
Code to execute when a user aborts an image load
Code to execute when an error occurs on image load
Code to execute on image load

Most of the attributes are optional; only a few are strictly necessary, as you can see in the following example:

<IMG NAME="dogImage"
SRC="images/dalmation.gif"
ALIGN="MIDDLE">
**HTML comment: <!- . . . ->**

Try to strike a balance between over-commenting and not adding enough comments to your code. Over-commenting can actually make your code harder to read; on the other hand, not adding any comments at all makes other HTML or JavaScript authors have to guess what your code is doing (and why the coding is doing it!).

```html
<!-
This HTML file was created by Juan Valdez,
01/01/97
->
</html>
```

Two types of comments exist: HTML comments, which are demonstrated in the preceding code, and JavaScript comments, which look like this: // or /* */. These comments are not interchangeable. If you try to use an HTML comment inside a JavaScript script, you get an error. If you try to use a JavaScript comment inside an HTML tag, your JavaScript comment shows up on the Web page!

For a good example of proper comment usage, see also “Getting Started with a Bare-Bones HTML Template,” in Part IX, or the “Comments” section in Part II.

**Frame: <FRAMESET> . . . <FRAME> . . .</FRAMESET>**

A frame is a special kind of window. You can have several frames per “regular” window, and a user can scroll each frame independently. Each frame can also be associated with a separate URL. What fun! What possibilities! (And — potentially — what confusion!)

To create a frame in HTML, use these tags:

```html
<FRAMESET> . . . <FRAME> . . . </FRAMESET>
```

To access that frame in JavaScript, use this identifier: frames[0]

In the preceding JavaScript identifier, 0 is a number representing the order in which the frame was defined in the HTML code. The first frame is represented by 0; the second, by 1; the third, by 2; and so on.

**Hyperlink anchor: <A> . . . </A>**

An anchor is a piece of text that uniquely identifies a spot on a Web page. After you define an anchor, you (or any other HTML author) can link to it.

To define an anchor in HTML, use this tag pair: `<A> . . . </A>`

To access that anchor in JavaScript, use this identifier: document.links[0]

In the preceding JavaScript identifier, 0 is a number representing the order in which the anchor was defined in the HTML code. The first anchor is represented by 0; the second, by 1; the third, by 2; and so on.
Inserting Form Elements into Your Web Page

You may notice that the `<A>...</A>` tag pair is used for both anchors and links. If you want, you can use one `<A>...</A>` tag pair to define a single piece of text that's both a link and an anchor.

### Tag Syntax for an Anchor

<table>
<thead>
<tr>
<th>Tag Syntax for an Anchor</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;A&gt;</code> Opening <code>&lt;A&gt;</code> tag</td>
<td></td>
</tr>
<tr>
<td>NAME=&quot;anchorName&quot;&gt; Name of anchor to which the links refers</td>
<td></td>
</tr>
<tr>
<td>anchorText Text to display at the anchor site</td>
<td></td>
</tr>
<tr>
<td><code>&lt;/A&gt;</code> Closing <code>&lt;A&gt;</code> tag</td>
<td></td>
</tr>
</tbody>
</table>

Here's a down-and-dirty anchor definition:

```
<A NAME="TOC">Table of Contents</A>
```

### Hypertext link: `<A>...</A>`

Hypertext links (or just plain links) are at the heart of the Web's usefulness. You can use links to connect and organize multiple pages of your own, to connect your pages with other Web pages, or both. Technically, a link is a piece of text (or an image) that loads another Web page when you click on it. (A link sometimes loads a specific spot, or anchor, on the other Web page.)

To define a hypertext link in HTML, use this tag pair: `<A>...</A>`

To access that link in JavaScript, use this identifier: `document.links[0]`

In the preceding JavaScript identifier, 0 is a number representing the order in which the link was defined in the HTML code. The first link is represented by 0; the second, by 1; the third, by 2; and so on.

### Tag Syntax for a Link

<table>
<thead>
<tr>
<th>Tag Syntax for a Link</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;A&gt;</code> Opening <code>&lt;A&gt;</code> tag</td>
<td></td>
</tr>
<tr>
<td>HREF=&quot;locationOrURL[#anchor]&quot; URL and (if appropriate) anchor to link to</td>
<td></td>
</tr>
<tr>
<td>[TARGET=&quot;windowName&quot;] Window to load linked page into (optional)</td>
<td></td>
</tr>
<tr>
<td>[onClick=&quot;handlerText&quot;] Code to invoke when a user clicks the link (optional)</td>
<td></td>
</tr>
<tr>
<td>[onMouseDown=&quot;handlerText&quot;] Code to invoke when mouse moves off the link (optional)</td>
<td></td>
</tr>
<tr>
<td>[onMouseOver=&quot;handlerText&quot;] Code to invoke when mouse moves across the link (optional)</td>
<td></td>
</tr>
<tr>
<td>linkText Text to display at link site</td>
<td></td>
</tr>
<tr>
<td><code>&lt;/A&gt;</code> Closing <code>&lt;A&gt;</code> tag</td>
<td></td>
</tr>
</tbody>
</table>

### Inserting Interactive Form Elements: `<INPUT>`

Many of the link's attributes are optional. You can use them all if you're feeling frisky, but the following is a good example of all that's really necessary to get the job done:

```
<A HREF="http://www.dummies.com">Link to the IDG Dummies Press Web Site</A>
```

When you move your mouse pointer over the link, the value specified for the HREF attribute (in this case, it's "http://www.dummies.com") appears in the status line at the bottom of the browser. You don't have to do anything special for this value to appear — the HTML interpreter does it for you automatically.

### Inserting Interactive Form Elements: `<INPUT>`

To define an interactive input element in HTML, use this tag:

```
<INPUT NAME="myInputElement"...>
```

To access that element in JavaScript, use this identifier:

```
document.myForm.myInputElement
```

All the elements in this section are form elements and so must be defined between the `<FORM>...</FORM>` tags.

The interactive elements that you can create in JavaScript are actually HTML elements, with one very important difference: JavaScript lets you add event handlers (onClick, onChange, onBlur, and so on) to elements so that the elements can respond to user interaction (see also Part VI). You can make several different types of input elements interactive with JavaScript:

<table>
<thead>
<tr>
<th>Input Element</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE=&quot;button&quot;</td>
<td>A customizable push button</td>
</tr>
<tr>
<td>TYPE=&quot;checkbox&quot;</td>
<td>Grouped check boxes (which allow multiple selections)</td>
</tr>
<tr>
<td>TYPE=&quot;file&quot;</td>
<td>A control that lets users browse and choose files</td>
</tr>
<tr>
<td>TYPE=&quot;hidden&quot;</td>
<td>A text element that users can't see</td>
</tr>
<tr>
<td>TYPE=&quot;password&quot;</td>
<td>A text element that displays characters as asterisks</td>
</tr>
<tr>
<td>TYPE=&quot;radio&quot;</td>
<td>Grouped radio buttons (which restrict users to one selection)</td>
</tr>
<tr>
<td>TYPE=&quot;reset&quot;</td>
<td>Predefined button to reset form values</td>
</tr>
<tr>
<td><code>&lt;SELECT&gt;</code></td>
<td>Configurable (single or multiple selection) list box</td>
</tr>
<tr>
<td>TYPE=&quot;submit&quot;</td>
<td>Predefined button to submit form values</td>
</tr>
<tr>
<td>TYPE=&quot;textarea&quot;</td>
<td>Single line text input field</td>
</tr>
<tr>
<td><code>&lt;TEXTAREA&gt;</code></td>
<td>Multiple line text input field</td>
</tr>
</tbody>
</table>
**Inserting Interactive Form Elements: <INPUT>**

The `<SELECT>` and `<TEXTAREA>` elements are input elements, too, but they're defined a little differently than the others. (Go figure.) Instead of identifying values for the `TYPE` attribute of the `<INPUT>` tag, they have tags all their own:

```
<SELECT>...</SELECT>
<TEXTAREA>...</TEXTAREA>
```

**Button**

A button is a clickable push button on an HTML form.

<table>
<thead>
<tr>
<th>Tag Syntax for a Button</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;INPUT&gt;</code> Single <code>&lt;INPUT&gt;</code> tag</td>
<td></td>
</tr>
<tr>
<td><code>TYPE=&quot;button&quot;</code> Specifies the kind of input control (button)</td>
<td></td>
</tr>
<tr>
<td><code>NAME=&quot;buttonName&quot;</code> Internal name of button (for coding purposes)</td>
<td></td>
</tr>
<tr>
<td><code>VALUE=&quot;buttonText&quot;</code> Text to display on face of button</td>
<td></td>
</tr>
<tr>
<td><code>[onClick=&quot;handlerText&quot;]</code> Code to invoke when a user clicks the button</td>
<td></td>
</tr>
</tbody>
</table>

Here's a real-life example:

```
<INPUT TYPE="button" NAME="infoButton" VALUE="Click Here for Info" onClick="displayInfo()" />
```

**Check box**

A check box is a toggle switch control. When users click a check box, they either check it (turn it on) or uncheck it (turn it off).

<table>
<thead>
<tr>
<th>Tag Syntax for a Check Box</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;INPUT&gt;</code> Single <code>&lt;INPUT&gt;</code> tag</td>
<td></td>
</tr>
<tr>
<td><code>TYPE=&quot;checkbox&quot;</code> Specifies the type of control (check box)</td>
<td></td>
</tr>
<tr>
<td><code>NAME=&quot;checkboxName&quot;</code> Internal name of check box (for coding purposes)</td>
<td></td>
</tr>
<tr>
<td><code>VALUE=&quot;checkboxValue&quot;</code> Value returned to the server when the form is submitted</td>
<td></td>
</tr>
<tr>
<td><code>[CHECKED]</code> Specifies initial display marked as checked (optional)</td>
<td></td>
</tr>
<tr>
<td><code>[onClick=&quot;handlerText&quot;]</code> Code to invoke when check box is clicked (optional)</td>
<td></td>
</tr>
<tr>
<td><code>textToDisplay</code> Descriptive text to display next to check box</td>
<td></td>
</tr>
</tbody>
</table>

Here's how you might go about creating a check box:

```
<INPUT TYPE="checkbox" NAME="classicalCheckbox" VALUE="classicalCheck" onClick="showClassicalTiltes()"> Click here if you like classical music.
```

**fileUpload**

A fileUpload element allows users to browse the file directories on their own machine and choose a file. Within a JavaScript script you can access the name of the chosen file with the `FILEUPLOADNAME.VALUE`, which you can then use however you see fit.

The fileUpload element is available only for Netscape Navigator 3.0 at the time of this writing.

<table>
<thead>
<tr>
<th>Tag Syntax for fileUpload</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;INPUT&gt;</code> Single <code>&lt;INPUT&gt;</code> tag</td>
<td></td>
</tr>
<tr>
<td><code>TYPE=&quot;file&quot;</code> Specifies the type of element (file)</td>
<td></td>
</tr>
<tr>
<td><code>NAME=&quot;fileUploadName&quot;</code> Specifies the name of the element</td>
<td></td>
</tr>
</tbody>
</table>

Here's what the definition of a fileUpload element looks like:

```
<INPUT TYPE="file" NAME="myFileUpload" />
```

**Hidden**

A hidden element is an input text field that doesn't appear on-screen. Hidden elements are usually used to store programmer-calculated values (behind-the-scenes program stuff) that get sent to the server when the user submits a form.

<table>
<thead>
<tr>
<th>Tag Syntax for a Hidden Element</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;INPUT&gt;</code> Single <code>&lt;INPUT&gt;</code> tag</td>
<td></td>
</tr>
<tr>
<td><code>TYPE=&quot;hidden&quot;</code> Specifies the type of element (hidden)</td>
<td></td>
</tr>
<tr>
<td><code>NAME=&quot;hiddenName&quot;</code> Internal name of hidden element (used in coding)</td>
<td></td>
</tr>
<tr>
<td><code>[VALUE=&quot;textValue&quot;]</code> Initial value of hidden object (optional)</td>
<td></td>
</tr>
</tbody>
</table>

Here's a garden-variety definition for a hidden element:

```
<INPUT TYPE="hidden" NAME="secretTextField" />
```
Password
A password object is a special text input field that displays asterisks on the screen (in place of the characters that the user actually types).

<table>
<thead>
<tr>
<th>Tag Syntax for a Password</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;INPUT</code> TYPE=&quot;password&quot;</td>
<td>Single &lt;INPUT&gt; tag</td>
</tr>
<tr>
<td>NAME=&quot;passwordName&quot;</td>
<td>Specifies the type of element (password)</td>
</tr>
<tr>
<td>SIZE=Integer</td>
<td>Internal name of password (for coding purposes)</td>
</tr>
<tr>
<td>[VALUE=&quot;textValue&quot;]</td>
<td>Number of characters to display initially</td>
</tr>
<tr>
<td></td>
<td>Initial value of password (optional)</td>
</tr>
</tbody>
</table>

Take a look at this sample password definition:
```
<INPUT TYPE="password" NAME="userPassword" SIZE=15 VALUE="secret!">
```

Radio button
A radio button is a toggle switch control, as is a check box. Unlike a check box, however, radio buttons are most often grouped in sets, which allow users to select a single option from a list.

<table>
<thead>
<tr>
<th>Tag Syntax for a Radio Button</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;INPUT</code> TYPE=&quot;radio&quot;</td>
<td>Single &lt;INPUT&gt; tag</td>
</tr>
<tr>
<td>NAME=&quot;radioName&quot;</td>
<td>Specifies the kind of element (radio)</td>
</tr>
<tr>
<td>VALUE=&quot;radioValue&quot;</td>
<td>Internal name of radio button (for coding purposes)</td>
</tr>
<tr>
<td>[CHECKED]</td>
<td>Specifies a value to be returned to the server</td>
</tr>
<tr>
<td>[onClick=&quot;handlerText&quot;]</td>
<td>Specifies that this button is initially selected (optional)</td>
</tr>
<tr>
<td>textToDisplay</td>
<td>Code to invoke when a user clicks the button (optional)</td>
</tr>
</tbody>
</table>

The following code shows a common way to define a set of radio buttons. Notice how the value for the NAME attribute is the same (timeChoice) for every radio button? Because all three radio buttons share one name (and only one value can be associated with one name at a time), the user is effectively restricted to selecting only one radio button from the entire group.

Reset
A reset object is a special kind of button. When a user clicks on the reset button, all the user-input values in a form are cleared and reset to their initial (default) values.

<table>
<thead>
<tr>
<th>Tag Syntax for a Reset Button</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;INPUT</code> TYPE=&quot;reset&quot;</td>
<td>Single &lt;INPUT&gt; tag</td>
</tr>
<tr>
<td>NAME=&quot;resetName&quot;</td>
<td>Specifies type of control (reset)</td>
</tr>
<tr>
<td>VALUE=&quot;buttonText&quot;</td>
<td>Internal name of control (for coding purposes)</td>
</tr>
<tr>
<td>[onClick=&quot;handlerText&quot;]</td>
<td>Text to display on face of reset button</td>
</tr>
<tr>
<td>textToDisplay</td>
<td>Code to invoke when a user clicks the button (optional)</td>
</tr>
</tbody>
</table>

Here's an example of how to add a reset button to your form:
```
<INPUT TYPE="reset" NAME="resetButton" VALUE="Reset Form Now" onClick="reinitializeFormulas()">
```

Select
The `<SELECT>` element is used to display both a single-selection list and a scrolling multiple-selection list box (depending on whether you specify the MULTIPLE attribute).

The `<SELECT>` element is one of the two input elements (`<TEXTAREA>` is the other) that don't follow the standard convention of specifying the element as the value for the `<INPUT>` tag's TYPE attribute. Unlike the other interactive objects that you put in an HTML document by using the `<INPUT>` tag and an attribute, you simply use the `<SELECT>` tag to put a list box in your Web page, as you can see in the following table.
### Inserting Interactive Form Elements: `<INPUT>`

<table>
<thead>
<tr>
<th>Tag Syntax for a Select List Box</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;SELECT&gt;</code></td>
<td>Opening <code>&lt;SELECT&gt;</code> tag</td>
</tr>
<tr>
<td>NAME=&quot;selectName&quot;</td>
<td>Internal name of control (for coding purposes)</td>
</tr>
<tr>
<td>[SIZE=integer]</td>
<td>Number of options visible (optional; default is 1)</td>
</tr>
<tr>
<td>[MULTIPLE]</td>
<td>Specifies multiple-selection scrolling box (optional)</td>
</tr>
<tr>
<td>[onBlur=&quot;handlerText&quot;]</td>
<td>Code to invoke when focus is lost (the user clicks elsewhere)</td>
</tr>
<tr>
<td>[onChange=&quot;handlerText&quot;]</td>
<td>Code to invoke when value changes and focus is lost</td>
</tr>
<tr>
<td>[onFocus=&quot;handlerText&quot;]</td>
<td>Code to invoke when focus is received (user clicks on element)</td>
</tr>
<tr>
<td><code>&lt;OPTION&gt;</code></td>
<td>Specifies a selection item</td>
</tr>
<tr>
<td>[VALUE=&quot;optionValue&quot;]</td>
<td>Value returned to server when user submits a form</td>
</tr>
<tr>
<td>[SELECTED]&gt;</td>
<td>Specifies that this option is selected by default</td>
</tr>
<tr>
<td>textToDisplay</td>
<td>Descriptive text to display next to option</td>
</tr>
<tr>
<td>&lt;/SELECT&gt;</td>
<td>Closing <code>&lt;SELECT&gt;</code> tag</td>
</tr>
</tbody>
</table>

You can repeat the `<OPTION>` tag (and its associated attributes) as needed, once for each option that you provide.

Because adding a list box to a Web page entails working not only with the usual associated attributes you’ve come to expect, but an additional tag, too (the `<OPTION>` tag), it can be a little confusing at first. Here’s a short, to-the-point example to help you get started:

```html
<select name="favoriteMusic" size=3 multiple
    onBlur="displayResult(this)"
>
  <option value="popChosen" selected> pop
  <option value="rockChosen"> rock
  <option value="dogsChosen"> dogs barking Jingle Bells
  <option value="rapChosen"> rap
  <option value="showChosen"> show tunes
  <option value="africanChosen"> African classical
</select>
```

### Submit

A `Submit` element is a special kind of button that submits (or sends) values from a form to a server (specifically, to a particular CGI program on a particular server that you define as part of the `<FORM>. . . </FORM>` tag pair). The submit button works in tandem with the `ACTION` attribute of the `<FORM>. . . </FORM>` tag pair. You can find more information about submitting form values to a CGI server in “Specifying Useful Web Page Features,” later in this part.

<table>
<thead>
<tr>
<th>Tag Syntax for a Submit Button</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;INPUT&gt;</code></td>
<td>Single <code>&lt;INPUT&gt;</code> tag</td>
</tr>
<tr>
<td>TYPE=&quot;submit&quot;</td>
<td>Specifies kind of control (submit)</td>
</tr>
<tr>
<td>NAME=&quot;submitName&quot;</td>
<td>Internal name of submit button (for coding purposes)</td>
</tr>
<tr>
<td>VALUE=&quot;submitText&quot;</td>
<td>Text to display on face of submit button</td>
</tr>
<tr>
<td>[onClick=&quot;handlerText&quot;]</td>
<td>Code to invoke when user clicks submit button (technically optional, but practically necessary!)</td>
</tr>
</tbody>
</table>

Here’s a practical example:

```html
<input type="submit" name="submitButton"
    value="Submit Form" onclick="verifyInput()"
>
```

### Text

The `text` element is a single-line input field.

<table>
<thead>
<tr>
<th>Tag Syntax for a Text Element</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;INPUT&gt;</code></td>
<td>Single <code>&lt;INPUT&gt;</code> tag</td>
</tr>
<tr>
<td>TYPE=&quot;text&quot;</td>
<td>Specifies the kind of control (text)</td>
</tr>
<tr>
<td>NAME=&quot;textName&quot;</td>
<td>Internal name of the text field (for coding purposes)</td>
</tr>
<tr>
<td>VALUE=&quot;textValue&quot;</td>
<td>Specifies initial value of text field</td>
</tr>
<tr>
<td>SIZE=integer</td>
<td>Number of characters to display before scrolling</td>
</tr>
<tr>
<td>[onBlur=&quot;handlerText&quot;]</td>
<td>Code to invoke when a user clicks elsewhere and focus is lost (optional)</td>
</tr>
<tr>
<td>[onChange=&quot;handlerText&quot;]</td>
<td>Code to invoke when value changes and focus is lost (optional)</td>
</tr>
</tbody>
</table>

(continued)
Inserting Interactive Form Elements: `<INPUT>`

<table>
<thead>
<tr>
<th>Tag Syntax for a Text Element</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[onFocus=&quot;handlerText&quot;]</code></td>
<td>Code to invoke when user clicks on the element and focus is received (optional)</td>
</tr>
<tr>
<td><code>[onSelect=&quot;handlerText&quot;]</code></td>
<td>Code to invoke when text in the field is highlighted (optional)</td>
</tr>
</tbody>
</table>

Here's a simple illustration of the text element in action:

```html
<INPUT TYPE="text" NAME="lastName" VALUE="Your name here" SIZE=30 onChange="validate()"/>
```

**Textarea**

A `<TEXTAREA>` element is just like a text element, except that instead of defining one scrolling input line, the `<TEXTAREA>` element defines a multline scroll box so that users can type in whole reams of text. Like `<SELECT>`, `<TEXTAREA>` doesn't follow the standard input element convention of specifying the element as the value for the `<INPUT>` tag's `TYPE` attribute. Instead, the `<INPUT>` tag is replaced in this case by the `<TEXTAREA>` tag, as shown in the following table.

<table>
<thead>
<tr>
<th>Tag Syntax for a Textarea Element</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;TEXTAREA&gt;</code></td>
<td>Opening <code>&lt;TEXTAREA&gt;</code> tag</td>
</tr>
<tr>
<td><code>NAME=&quot;textareaName&quot;</code></td>
<td>Internal name of textarea object (for coding)</td>
</tr>
<tr>
<td><code>ROWS=integer</code></td>
<td>Number of rows to display</td>
</tr>
<tr>
<td><code>COLS=integer</code></td>
<td>Number of columns to display</td>
</tr>
<tr>
<td><code>[onBlur=&quot;handlerText&quot;]</code></td>
<td>Code to invoke when textarea loses focus (optional)</td>
</tr>
<tr>
<td><code>[onChange=&quot;handlerText&quot;]</code></td>
<td>Code to invoke when value changes and focus is lost (optional)</td>
</tr>
<tr>
<td><code>[onFocus=&quot;handlerText&quot;]</code></td>
<td>Code to invoke when focus is received (optional)</td>
</tr>
<tr>
<td><code>[onSelect=&quot;handlerText&quot;]</code></td>
<td>Code to invoke when text is highlighted (optional)</td>
</tr>
<tr>
<td><code>textToDisplay</code></td>
<td>Initial text to display inside scroll box</td>
</tr>
<tr>
<td><code>&lt;/TEXTAREA&gt;</code></td>
<td>Closing <code>&lt;TEXTAREA&gt;</code> tag</td>
</tr>
</tbody>
</table>

Inserting a Java Applet: `<APPLET>`...

Here's a garden-variety textarea example for you to see:

```html
<TEXTAREA NAME="directions" ROWS=4 COLS=60
onBlur="validate()" onChange="display()"
onFocus="welcome()" onSelect="changeMode()">
This is default text. You can type right over it, add to it, cut it, paste it, or copy it.
</TEXTAREA>
```

**Inserting a Java Applet:** `<APPLET>`...

A Java applet is a special little software program specifically designed to live inside a Web page.

*JavaScript For Dummies*, by yours truly, contains a whole chapter dedicated to Java and JavaScript-to-Java interaction. Also check out *Java For Dummies*, by Aaron Walsh, for more low-down on Java applets.

To embed a Java applet in your Web page, use this tag pair:

```html
<APPLET>...</APPLET>
```

To access a Java applet in JavaScript, use this identifier:

```javascript
document.applets[0]
```

In the preceding JavaScript identifier, 0 is a number representing the order in which the applet was defined in the HTML code. The first applet you define is represented by 0; the second, by 1; the third, by 2; and so forth.

When you embed a Java applet in your HTML code, you need to know a little bit about that applet. For example, some applets require `parameters`, which are values that you send an applet when you embed it. (Because applet function is restricted only by the Java programmer's imagination, the parameters you may be required to supply are equally diverse.) If you want to embed an applet that requires parameters, you have to include those parameters in your `<APPLET>` definition. Two ways exist to find out what any given applet requires so that you can embed it in your Web page:

- Surf the Web until you find an example of how someone else embedded the applet and follow that example. (Typically, this is how you find out about applets in the first place.)
- E-mail the applet programmer directly and ask for instructions.
Tag Syntax to Embed a Java Applet | Explanation
---|---
`<APPLET>` | Beginning `<APPLET>` tag
CODE="classFileName" | File name of the applet to load (*.class)
HEIGHT=height | Height of the applet, in pixels
WIDTH=width | Width of the applet, in pixels
[MAYSCRIPT] | If specified, this attribute permits the applet to access JavaScipt values (optional)
[NAME="appletName"] | Name of applet for internal coding purposes (optional but recommended)
[CODEBASE="classFileDirectory"] | Directory where classFileName is located (optional but recommended)
[ALT="alternateText"] | Text to display in place of applet (for browsers that don't support Java; optional)
[ALIGN="position"] | Specifies alignment of applet on HTML page (optional)
[HSPACE=numberOfPixels] | Horizontal margin of applet, in pixels (optional)
[VSPACE=numberOfPixels] | Vertical margin for applet, in pixels (optional)

You can repeat the following optional section as many times as necessary:

```
[<PARAM>
NAME="parameterName" | Name of parameter to pass to applet
VALUE="parameterValue">
</PARAM>
```

The value for ALIGN can be any of the following string values: LEFT, RIGHT, TOP, ABSMIDDLE, ABSBOTTOM, TEXTTOP, MIDDLE, BASELINE, or BOTTOM.

Here's an example of code that embeds an applet that doesn't require any parameters. (It actually works, if you want to try it out for yourself; if you do, you'll see a handful of animated, multicolored bubbles drift lazily from the bottom of the applet space up to the top.)

```html
<br>
<APPLET CODE="BubbleApplet"
CODEBASE="http://java.sun.com/applets/applets/Bubbles"
WIDTH=500 HEIGHT=500>
</APPLET>
```

Mapping an Area: `<MAP>...<AREA>...</MAP>`

An area is like a big, thick link: An area describes a space on your page, which you can shape and size as you want and which responds to user events. Areas are used to create what are called clickable maps; as users move their mouse around on an image, the link (or URL) that appears in the user's Web status line changes.

To embed an area in your Web page, use this tag pair:

```
<MAP>...<AREA>...</MAP>
```

To access that area in JavaScript, use this identifier:

document.links[0]

No, the preceding line isn't a typo! Areas are stored in the links array, right along with regular, "skinny" links.

Tag Syntax to Embed an Area | Explanation
---|---
`<MAP>` | Opening `<MAP>` tag (you need to define a map to define an area)
NAME="mapName" | Name of map for internal coding purposes
`<AREA>` | Opening `<AREA>` tag
NAME="areaName" | Name of area for internal coding purposes
COORDS="x1, y1, x2, y2" | Coordinates of the image map, in integers
HREF="location" | URL of document to load when user clicks on an area, or "javascript:functionName()"
[SHAPE="shape"] | Shape of the map (see the Tip that follows the table)
[TARGET="windowName"] | Window to load URL into when a user clicks on an area
 onMouseOut="handlerText"] | Code to execute when mouse pointer moves out of area
 onMouseOver="handlerText"] | Code to execute when mouse pointer is dragged across area
</MAP> | Closing `<MAP>` tag
Plugging in a Plug-in <EMBED>... </EMBED>

Here's a sample area definition:

```html
<MAP NAME="thistleMap">
  <AREA NAME="topThistle" COORDS="0.0,0.228,318" HREF="javascript:void(0)"
    onMouseOver="self.status='That mouse pointer sure feels nice'; return true"
    onMouseOut="self.status='Thanks for visiting; come again!'; return true">
</MAP>
```

The preceding code snippet creates a map that matches the boundaries of a picture. (The picture in this case happens to be of bright pink thistles, which is why the name of the map is thistleMap.) When a user moves the cursor onto the picture/map, the text 'That mouse pointer sure feels nice' appears in the status bar. When a user drags the cursor away from the picture/map so that the cursor rests somewhere else on the screen, the text in the status bar changes to "Thanks for visiting; come again!"

The value for SHAPE can be any of the following strings: "rect", "poly", "circle", or "default". If you don't define any value for the SHAPE attribute, the shape defaults to "rect".

Plugging in a Plug-in: <EMBED>... </EMBED>

Netscape plug-ins are software components that "plug in" to Netscape Navigator to extend its capabilities.

To embed a plug-in into your Web page, use this tag pair:

```html
<EMBED>...</EMBED>
```

To access that plug-in in JavaScript, use this identifier:

```javascript
navigator.plugins[0]
```

In the preceding JavaScript identifier, 0 is a number representing the order the plug-in was embedded in your HTML code. The first plug-in you embed is represented by 0; the second, by 1; the third, by 2; and so forth.

You define the plug-in object by using the <EMBED> tag. The output of the plug-in that you embed appears in the same space on your form where the plug-in object appears.

Specifying Useful Web Page Features

<table>
<thead>
<tr>
<th>Tag Syntax to Embed a Plug-In</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;EMBED&gt;</code></td>
<td>Opening <code>&lt;EMBED&gt;</code> tag</td>
</tr>
<tr>
<td><code>SRC=&quot;source&quot;</code></td>
<td>URL containing the source of the plug-in</td>
</tr>
<tr>
<td><code>NAME=&quot;embedName&quot;</code></td>
<td>Name of embedded plug-in object (for internal coding purposes)</td>
</tr>
<tr>
<td><code>HEIGHT=height</code></td>
<td>Height of embedded plug-in, in pixels</td>
</tr>
<tr>
<td><code>WIDTH=width</code></td>
<td>Width of embedded plug-in, in pixels</td>
</tr>
<tr>
<td>You can repeat the following optional section as many times as necessary:</td>
<td></td>
</tr>
<tr>
<td><code>[&lt;PARAM&gt;</code></td>
<td>Opening <code>&lt;PARAM&gt;</code> tag</td>
</tr>
<tr>
<td><code>NAME=&quot;parameterName&quot;</code></td>
<td>Name of parameter (argument) to pass to plug-in</td>
</tr>
<tr>
<td><code>VALUE=&quot;parameterValue&quot;&gt;</code></td>
<td>Value of parameter (argument) to pass to plug-in</td>
</tr>
<tr>
<td><code>[/PARAM&gt;]</code></td>
<td>Closing <code>&lt;/PARAM&gt;</code> tag</td>
</tr>
<tr>
<td><code>&lt;EMBED&gt;</code></td>
<td>Closing <code>&lt;/EMBED&gt;</code> tag</td>
</tr>
</tbody>
</table>

If you're unfamiliar with plug-ins — what they are, how they work, and why the heck you'd ever want to embed one in your Web page — you may want to take a look at the following URL which lists a treasure trove of plug-ins for sale:

```
http://home.netscape.com/comprod/
development_partners/plugin_api/
plugin_design.html
```

Here's an example that you can try out for yourself. This code uses the Adobe Acrobat plug-in to display a .pdf file, which is a file with a fancy graphic display format.

```html
<EMBED NAME="myEmbed" SRC="http://
  ecco.bsee.swin.edu.au/text/adobe/PDFs/
  AcroCD.pdf"
  WIDTH=450 HEIGHT=450>
</EMBED>
```

Specifying Useful Web Page Features

This section contains examples of useful things you may want to configure your Web pages to do:

- Call a CGI program
- Display a title
Specifying Useful Web Page Features

CGI program to call on submit: <FORM>...<FORM>

When a user clicks on a submit push button in a form, all the values contained in that form are sent automatically to a Common Gateway Interface (CGI) program. You specify a particular CGI program as the value of the ACTION attribute of the <FORM> tag. If you don’t specify a CGI program for the ACTION attribute, nothing happens when a user clicks on the submit push button, which is great for testing. In real life, though, if you’re going to the trouble of adding a submit push button, you need to specify a value for the ACTION attribute, as shown in the following code:

```
<FORM
    ACTION="http://www.madeup.com/cgi-bin/someCGIProgram"
>
</FORM>
```

Wait until your Web page is behaving nicely before you specify a CGI program to which you want to submit your form data. That way, you can test and refine your JavaScript statements first (before you complicate matters by introducing the CGI program).

Title for your Web page: <TITLE>...</TITLE>

Titles play an important part in communicating your message (whatever that may be) to your users. Not only are the words in your title used by many Web-searching programs, but when users save a reference to your Web page (sometimes called bookmarking), your title is what appears on their list of saved references. For these reasons alone it’s worth spending a little time on wording your title.

Notice in the following example that the <TITLE>...</TITLE> tags are placed between the <HEAD>...</HEAD> tags.

```
<HTML>
    <HEAD>
        <TITLE>
            Dave's Retail Catalogue of Restored Antique Wood-Burning Stoves
        </TITLE>
    </HEAD>
</HTML>
```

Although you’re not required to place the title between the <HEAD> tags (the title will still appear if you place the <TITLE>...</TITLE> tags after the <HEAD>...</HEAD> tags), the example shows the standard style used by the Web-savvy crowd.

JavaScript Basics

This part is like a JavaScript grammar book, dictionary, and thesaurus— all rolled into one. In this part, you find the nuts-and-bolts mechanics of writing JavaScript statements — from syntax to special keywords, from declaring variables to defining and calling functions. You also find an overview of all the objects you can work with in JavaScript. (For a detailed description of each object in the overview, see also Part III.)

In this part...

- Understanding the security issues associated with JavaScript scripts
- Becoming familiar with the JavaScript object model
- Unraveling JavaScript syntax and expressions
About JavaScript Security

Because JavaScript runs on the client computer, its ability to cause security breaches is fairly limited. Security issues are more of a concern when client/server communication is involved, and this type of communication isn't included in JavaScript's bag of tricks.

As of this writing, only one minor problem has been widely publicized, and that's the ability of a mischievous JavaScripter to set up a harmless-looking button that sends an e-mail message (complete with your e-mail address) to the mischief-maker, without your knowledge.

Fortunately, this problem only rears its ugly head with the Netscape Navigator 3.0 Web browser, and it's completely preventable. From the Navigator browser main window, all you have to do is choose Options → Network Preferences → Protocols. Then, in the Show an Alert Before box, click the check box labeled Submitting a Form by Email, and the problem is solved: Any time that your Web browser attempts to send e-mail, an alert window notifies you and gives you the opportunity to cancel the proceedings.

Because developers tend to take security issues very seriously, expect each version of Netscape Navigator and Microsoft Internet Explorer to be more secure than its predecessor.

Web security is a hot topic these days — whenever information passes across the Internet, there's always an associated security hazard. CGI programs, plug-ins, Java applets, and cookies all present different security risks that you may want to be aware of. If you plan to integrate your JavaScript scripts with any of these elements (or just to find out more about staying secure on the Internet, in general), check out Computer Security For Dummies, by Peter Davis and Barry Lewis (published by IDG Books Worldwide, Inc.).

This Web site is devoted exclusively to JavaScript-related security issues:

http://ciac.llnl.gov/ciac/javasecure.html

Basics of the JavaScript Object Model

You can work with three main kinds of objects in JavaScript:
- Built-in data types
- Objects that make up a Web page
- Utility objects

Read on for a quick rundown of each object type and the differences between them.

Built-in JavaScript data types

Numbers, Boolean values (true or false), and strings (a bunch of characters surrounded by quotes, like "this") are such basic programming building blocks (called data types) that you don't even have to create special objects to use them in JavaScript. All you have to do is specify a numeric, Boolean, or string value, and the JavaScript interpreter takes care of the rest.

Look at the following examples to see what I mean:

<table>
<thead>
<tr>
<th>Built-in Data Type</th>
<th>JavaScript Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>var loveWork = true</td>
</tr>
<tr>
<td>null</td>
<td>var middleInitial = null</td>
</tr>
<tr>
<td>number</td>
<td>var myAge = 29</td>
</tr>
<tr>
<td>string</td>
<td>var fullName = &quot;Kris Kringle&quot;</td>
</tr>
</tbody>
</table>

In the preceding table, you create four different variables to hold four different values, each associated with a different data type. The first variable, loveWork, is assigned the Boolean value true; the second variable, middleInitial, is assigned the null value; the third variable, myAge, is assigned a number; and the fourth value, fullName, is assigned a string value. Taken together, the JavaScript interpreter reads these variables as, "Kris Kringle is age 29, has no middle initial, and loves his work."

The null data type means "nothing" (which is different from simply not assigning any value). The null data type is a valid value all on its own.

Hierarchy of JavaScript objects

You can think of the JavaScript object hierarchy as your favorite browser's object hierarchy, because you create these objects in HTML (a few are even created for you automatically by the browser itself) before JavaScript ever enters the picture. After the objects exist, JavaScript lets you examine them, change them, perform calculations based on them, and do pretty much whatever else your heart desires.

Notice in the following table that when one object is contained in another, that containment is reflected in the JavaScript syntax. For example, a button is defined in HTML as part of a form which, itself, is part of the overall HTML document. So if you want to access a button, you need to type this code:

document.nameOfYourForm.nameOfYourButton
<table>
<thead>
<tr>
<th>Object</th>
<th>HTML Tag</th>
<th>JavaScript Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>window</td>
<td>none (it's a given)</td>
<td>window (optional)</td>
</tr>
<tr>
<td>document</td>
<td>&lt;HTML&gt;...&lt;/HTML&gt;</td>
<td>document</td>
</tr>
<tr>
<td>anchor</td>
<td>&lt;A&gt;...&lt;/A&gt;</td>
<td>document.links[0]</td>
</tr>
<tr>
<td>applet</td>
<td>&lt;APPLET&gt;...&lt;/APPLET&gt;</td>
<td>document.applets[0]</td>
</tr>
<tr>
<td>area</td>
<td>&lt;MAP&gt;...&lt;/AREA&gt;...&lt;MAP&gt;</td>
<td>document.someArea</td>
</tr>
<tr>
<td>form</td>
<td>&lt;FORM&gt;...&lt;/FORM&gt;</td>
<td>document.someForm</td>
</tr>
<tr>
<td>button</td>
<td>&lt;INPUT TYPE=&quot;button&quot;&gt;</td>
<td>document.someForm.someButton</td>
</tr>
<tr>
<td>checkbox</td>
<td>&lt;INPUT TYPE=&quot;checkbox&quot;&gt;</td>
<td>document.someForm.myCheckbox</td>
</tr>
<tr>
<td>fileUpload</td>
<td>&lt;INPUT TYPE=&quot;file&quot;&gt;</td>
<td>document.someForm.myFileUpload</td>
</tr>
<tr>
<td>hidden</td>
<td>&lt;INPUT TYPE=&quot;hidden&quot;&gt;</td>
<td>document.someForm.someHidden</td>
</tr>
<tr>
<td>image</td>
<td>&lt;IMG&gt;</td>
<td>document.someForm.someImage</td>
</tr>
<tr>
<td>password</td>
<td>&lt;INPUT TYPE=&quot;password&quot;&gt;</td>
<td>document.someForm.somePassword</td>
</tr>
<tr>
<td>radio</td>
<td>&lt;INPUT TYPE=&quot;radio&quot;&gt;</td>
<td>document.someForm.someRadio</td>
</tr>
<tr>
<td>reset</td>
<td>&lt;INPUT TYPE=&quot;reset&quot;&gt;</td>
<td>document.someForm.someReset</td>
</tr>
<tr>
<td>select</td>
<td>&lt;SELECT&gt;...&lt;/SELECT&gt;</td>
<td>document.someForm.someSelect</td>
</tr>
<tr>
<td>submit</td>
<td>&lt;INPUT TYPE=&quot;submit&quot;&gt;</td>
<td>document.someForm.someSubmit</td>
</tr>
<tr>
<td>text</td>
<td>&lt;INPUT TYPE=&quot;text&quot;&gt;</td>
<td>document.someForm.someText</td>
</tr>
</tbody>
</table>

**See also** Part I to find out about the individual HTML tags that create each object.

Both the `window` and the `frame` objects have associated aliases. (An alias is an alternative way of referring to an object and should be used if doing so makes your code easier to understand.)

You can refer to a `window` by any of the following identifiers: `parent` (if the window in question is the parent of the window containing the reference); `self` (if the window in question is the same window as the one containing the reference); or `top` (if the window in question is at the top of the window hierarchy containing the reference).

Similarly, a `frame` can be referred to by either `parent` or `self`.

**Utility objects**

Utility objects, unlike the objects that make up the JavaScript object hierarchy, don't represent any one piece of a Web page. As the following table shows, utility objects are just useful stand-alone utilities — arrays, date, functions — that you may want to use in your JavaScript statements.

<table>
<thead>
<tr>
<th>Utility Object</th>
<th>JavaScript Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>var myPets = new Array (&quot;Spike&quot;, &quot;Zeke&quot;, &quot;Fluffy&quot;)</td>
</tr>
<tr>
<td>Date</td>
<td>var today = new Date()</td>
</tr>
<tr>
<td>Function</td>
<td>var salary = new Function (&quot;base&quot;, &quot;commission&quot;, &quot;return base + (base * commission)&quot;)</td>
</tr>
</tbody>
</table>

(continued)
### Basic Punctuation and Syntax

<table>
<thead>
<tr>
<th>Utility Object</th>
<th>JavaScript Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math</td>
<td>var randomNumber = Math.random();</td>
</tr>
<tr>
<td>Option</td>
<td>var blues = new Option(&quot;Blues music&quot;, &quot;blues&quot;, true, true);</td>
</tr>
</tbody>
</table>

An **Option** object isn’t much use by itself; it’s only meaningful when it’s related to a **select** list box. Normally, you define options with HTML statements at the same time you define the list box itself. Alternatively, you can define them as shown with the **Option** utility object.

### Basic Punctuation and Syntax

As programming languages go, JavaScript is pretty easy to learn. (I know, easy for me to say!) Just like English, the JavaScript language is made up of words and punctuation, which you, gentle JavaScripter, must combine to form meaningful statements. Between Part III, which describes all the JavaScript objects in detail, and this part, which describes all the JavaScript syntax and keywords, you have everything you need to write your very own scripts.

Some JavaScript interpreters are a little more forgiving than others, but no guarantee exists that future versions won’t tighten the screws a bit. What that means is that, while bending the punctuation rules in this section may work for now (for example, you may be able to get away with leaving off a piece of punctuation here or there), it probably won’t work in all browsers, or for very long (and long in Web years is about three months!). To be on the safe side, always follow the guidelines in this section.

If you are familiar with C or C++, you may immediately notice JavaScript’s lack of a statement-ending semicolon. Punctuation is a little less complicated in JavaScript than it is in C or C++, but it’s not foolproof! JavaScript still provides you the flexibility to make a couple of annoying punctuation errors — and this section helps you avoid them.

### Nested quotes

You use quotes in JavaScript — both single quotes (’’) and double quotes (""") — to surround string values. Why both kinds of quotes? Because you may run into a situation where you need to use two sets of quotes in a single JavaScript statement. If so, you need to use both single and double quotes and alternate them.

### If you try to nest double quotes inside double quotes (or single quotes inside single quotes), you run into trouble. Here’s an example:

```javascript
onClick="alert('This is an example of correctly nested quotes.')"
```

```javascript
onClick="alert("Warning! This statement will produce an error.")"
```

```javascript
onClick='alert('Warning! This statement is wrong, too.\')'
```

If you want a double quote to appear in a string, here’s what you do: Precede the double quote with a backslash. (This action is called *escaping* the quote.) Here’s what it looks like:

```javascript
alert("Did you like the movie "Phenomenon"?")
```

### Pairs

JavaScript scripts are typically rife with pairs — pairs of opening and closing tags and angle brackets (courtesy of HTML), pairs of parentheses, pairs of quotes, and pairs of curly braces. If you forget to add a closing bracket, brace, or whatever, the JavaScript interpreter complains. Sometimes the complaint takes the form of a syntax error; sometimes you get a goofy-looking page display.

Following are some examples of pair mismatching to look out for:

<table>
<thead>
<tr>
<th>HTML/JavaScript Statement</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;FORM NAME=myForm</code></td>
<td></td>
</tr>
<tr>
<td>Missing right angle bracket (&gt;)</td>
<td></td>
</tr>
<tr>
<td><code>&lt;A HREF=&quot;http://www.tucows.com&quot;&gt;Two Cows</code></td>
<td></td>
</tr>
<tr>
<td>Missing closing tag (<code>&lt;/A&gt;</code>)</td>
<td></td>
</tr>
<tr>
<td>alert(&quot;Form processing complete.&quot;)`</td>
<td></td>
</tr>
<tr>
<td>Missing parenthesis ()</td>
<td></td>
</tr>
<tr>
<td><code>firstName = &quot;Barney</code></td>
<td></td>
</tr>
<tr>
<td>Missing quote (&quot;&quot;&quot;)</td>
<td></td>
</tr>
<tr>
<td><code>if (name == &quot;&quot;) { alert(&quot;Please enter your name.&quot;</code></td>
<td></td>
</tr>
<tr>
<td>Missing curly brace (})</td>
<td></td>
</tr>
</tbody>
</table>

### Spelling and capitalization (case)

All the words you use in programming JavaScript must be spelled correctly. For example, if you create a variable called `lastName` and then try to display it on your Web page but misspell it as `lastNam`, you get an error. As close as these two words may appear to human eyes, they look nothing alike to the JavaScript interpreter.
Character case is just as important as correct spelling. For example, the JavaScript interpreter won't recognize the variable named `lastName` if you type it `LastName`.

**Top-down**

The JavaScript interpreter reads from top to bottom, left to right. So, before you can use something, that something must first be defined. Case in point: In order to call (or use) a function, you must first define that function in an earlier statement. Likewise, if you want to access a variable, you must declare that variable first.

### Comments (/*...*/ and //)

Comments aren't processed at all by the JavaScript interpreter; they're ignored. A comment's purpose is to give script authors a free-form way to communicate with themselves (you'd be surprised how quickly you forget why you did something the way you did it) and any other humans who read their scripts.

Two different kinds of JavaScript comments exist. Either can appear anywhere in your script, as many times as you want.

You create a single-line comment by typing a double slash (`//`) at the beginning of the line, followed by your comment, like this:

```javascript
// This is a single-line comment.
```

Create a multiple-line comment by beginning a line with `/*` and ending your comment with `*/`, like so:

```javascript
/*
This comment can span multiple lines. Always remember to close it off, though; if you forget, you'll get weird errors when you try to display your script. */
```

Nesting multiple-line comments is a bad idea. A block of code like the following can cause grief because the interpreter ignores the second `/*` when it gets to the first `*/`:

```javascript
/* Blocking out this section for testing purposes... */
/* Here is a comment. */
```

---

### Conditional Expressions: if . . . else

The `if...else` expression is called a *conditional* expression because you use it to test whether a certain condition is true. A condition can be a variable, a statement, or an expression — anything at all that can be resolved by the JavaScript interpreter to a simple true or false answer.

If the condition is `true`, the interpreter executes all the statements between curly braces that follow the `if` clause. If the condition is `false`, the interpreter executes all the statements between curly braces that follow the `else` clause. Here's the generic description of how to use `if...else`:

```javascript
if (condition) {
    statements
} else {
    statements
}
```

The square brackets around the `else` clause mean that the `else` clause is optional — it's possible to code just the `if` clause, if you want. And no rule says that an `if...else` expression can't have other statements nested inside of it, either (many do). Just remember to include the curly braces as shown for each `if...else`. There's no leeway here; they have to be curly braces, not parentheses, and they have to come in pairs, just like in the following example:

```javascript
if (numberOrdered <= 100) {
    // calculate the order at retail cost
    calculateTotal(19.95)
} else {
    // calculate the order at wholesale cost
    calculateTotal(11.00)
}
```

---

### Loops

Loops are common programming constructs that you can use to perform a single task many times, in as compact a way as possible. JavaScript contains two basic kinds of loops: `for` and `while`. Both types of loops are explained in the following sections, along with some other keywords that you can use with `for` and `while` to create concise, powerful loops.
break

The break keyword must be used inside of a loop (your loop choices are for, for...in, and while). When the JavaScript interpreter encounters a break statement, it breaks out of the loop entirely and starts interpreting again at the first line following the loop. For example:

```
for (var i = 1; i <= 20; i++) {
    if (i == 13) { // Only go up to 12
        break
    }
    document.writeln(i)
}
```

// this is where the interpreter will pick up again
// after the break

Here's how the output will look:
1 2 3 4 5 6 7 8 9 10 11 12

continue

Like break, continue can be used inside for, for...in, or while loops. When the JavaScript interpreter encounters a continue statement, it stops what it's doing and hops back up to the beginning of the loop to continue as normal. The following example may make it clearer:

```
for (var i = 1; i <= 20; i++) {
    if (i == 13) { // Superstitious! Don't print #13
        continue
    }
    document.writeln(i)
}
```

The following output shows you exactly how continue works.
You may want to compare the following output to the output generated by break:
1 2 3 4 5 6 7 8 9 10 11 12 14 15 16 17 18 19 20

In the output generated by continue, you can see that the number 13 is skipped — but then the loop continues and prints out the numbers 14 through 20 (unlike the break command, which stops the loop dead in its tracks after printing the number 12).

The continue keyword is useful for handling exceptions to a rule. For example, you may want to process all the items in a group the same way except for one or two special cases.

for

The for loop comes straight from the C language — and since C is famous for its terseness, it won't come as a shock to you that in the wrong hands, for loops can be positively Byzantine.

First, have a look at the generic form:

```
for ([initial expression]; [condition]; [update expression]) {
    statements
}
```

The preceding syntax introduces three terms that may be new to you:

- **Initial expression**: Think of the initial expression as the starting point — a snapshot of how things look right before the interpreter hops into the loop and gets down to business.
- **Condition**: The condition is the JavaScript expression to be tested each time the interpreter takes a pass around the loop.
- **Update expression**: If the condition tests true, the JavaScript interpreter performs the update expression before looping around to test the condition again.

Here's a short example that should help make the workings of the for loop crystal clear:

```
for (var i = 1; i <= 10; i++) {
    document.writeln(i)
        
```

The following steps describe what happens in the preceding for loop:

1. The variable i is set equal to 1.
2. The JavaScript interpreter checks to see whether i is less than or equal to 10.
3. i is less than or equal to 10, so the body of the loop executes.
4. i is written to the screen (this one action, document.writeln(i), forms the entire body of the loop).
5. The JavaScript interpreter adds 1 to i; now i is 2.
6. The JavaScript interpreter checks to see whether i is less than or equal to 10.
7. i is less than or equal to 10, so the body of the loop executes.
8. i is written to the screen via the document.writeln method (again, this one action comprises the entire loop body).
9. The JavaScript interpreter adds 1 to i; now i is 3.
10. Start again at Step 6.

See the pattern? The interpreter begins at the top of the loop and performs the body of the loop once for each time that the loop condition is true.

**for ... in**

If you like the for loop, you'll love for...in. The two are very similar, but unlike for, you use the for...in loop exclusively for looping, or *iterating*, through all the properties of an object, like so:

```
for (var in object) {
  statements
}
```

To elaborate, here's a function that loops through all the properties of a given object. As the function loops through, it builds a string (called result) containing the name and value of each property it finds.

```
function displayProperties(inputObject, inputPropertyName) {
  var result = "";
  for (var eachProperty in inputObject) {
    result += " 
    + eachProperty
    + " = "
    + inputObject[eachProperty]
    + "<BR>"
  }
  result += "<HR>"
  return result
}
```

Here's what the output for the preceding code snippet might look like if the text input element aTextField were passed into the loop:

```
document.myForm.aTextField.type = text
document.myForm.aTextField.name = aTextField
document.myForm.aTextField.form = [object Form]
document.myForm.aTextField.value = My dog has fleas.
document.myForm.aTextField.defaultValue = null
```

**while**

The while loop is similar to the for loop. First you set up a condition, and while that condition is true, the JavaScript interpreter executes the statements in the body of the loop. If the condition is never true, the statements never execute; if the condition is always true, well, let's just say that those statements will execute for a long, long, long time. Obviously, then, you want to make sure that one of the statements in the body of your while loop changes the while condition in some way so that it stops being true at some point.

First, the generic syntax:

```
while (condition) {
  statements
}
```

Now here's an example of while in action:

```
while (totalInventory > numberPurchased) {
  totalInventory = totalInventory - numberPurchased
  numberSales++
}
```

**Operators**

Operators are kind of like conjunctions in English: You use operators to join multiple phrases together to form expressions. The operators you're familiar with in everyday life include the plus sign (+) and the minus sign (-). JavaScript provides you with a lot more operators, however, as you can see in the following sections.

**Assignment operators**

Assignment operators let you assign values to variables. Besides being able to make a straight one-to-one assignment, though, you can also use some of them as a kind of shorthand to bump up a value based on another value. The following table describes how each operator works:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = y</td>
<td>The value of y is assigned to x</td>
<td>x = 10, y = 15 each time</td>
</tr>
<tr>
<td>x += y</td>
<td>x = x + y (addition)</td>
<td>x = 25</td>
</tr>
<tr>
<td>x -= y</td>
<td>x = x - y (subtraction)</td>
<td>x = -5</td>
</tr>
<tr>
<td>x *= y</td>
<td>x = x * y (multiplication)</td>
<td>x = 150</td>
</tr>
<tr>
<td>x /= y</td>
<td>x = x / y (division)</td>
<td>x = .6666666</td>
</tr>
<tr>
<td>x %= y</td>
<td>x = x % y (modulus)</td>
<td>x = 10</td>
</tr>
</tbody>
</table>
Here's how the modulus operator works: \( x \% y \) means that the interpreter tries to divide \( y \) into \( x \) evenly. The result is anything left over. In the preceding example, \( x \) is 10 and \( y \) is 15. 15 won't go into 10 evenly at all, so 10 is what's left over.

**Comparison operators**

When comparing two values or expressions for equality, you've got the following choices:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{==} ) (two equal signs)</td>
<td>( x \text{ == } y )</td>
<td>( x ) is equal to ( y )</td>
</tr>
<tr>
<td>( \text{!=} )</td>
<td>( x \text{ != } y )</td>
<td>( x ) is not equal to ( y )</td>
</tr>
<tr>
<td>( \text{&lt;} )</td>
<td>( x \text{ &lt; } y )</td>
<td>( x ) is less than ( y )</td>
</tr>
<tr>
<td>( \text{&gt;} )</td>
<td>( x \text{ &gt; } y )</td>
<td>( x ) is greater than ( y )</td>
</tr>
<tr>
<td>( \text{&lt;=} )</td>
<td>( x \text{ &lt;= } y )</td>
<td>( x ) is less than or equal to ( y )</td>
</tr>
<tr>
<td>( \text{&gt;=} )</td>
<td>( x \text{ &gt;= } y )</td>
<td>( x ) is greater than or equal to ( y )</td>
</tr>
<tr>
<td>( \text{?:} )</td>
<td>( x = (y &lt; 0) \text{ ? } -y \text{ : } y )</td>
<td>If ( y ) is less than zero, assign (-y) to ( x); otherwise, assign ( y ) to ( x)</td>
</tr>
</tbody>
</table>

A common mistake that beginning programmers often make is using a single equal sign in place of a double equal sign (and vice versa). JavaScript doesn't complain if you do this; after all, both \( (x == 6) \) and \( (x = 6) \) are legal expressions, and JavaScript has no way of knowing which expression you really want to state. The two examples are radically different, though, and interchanging them can wreak havoc on your logic. The first example compares 6 to \( x \), and the second assigns 6 to \( x \)!

**Logical operators**

Logical operators work on logical values (also called Boolean values) and they also return Boolean values. A Boolean value can only be one of two things: It's either true or it's false. When you see two expressions separated by a logical operator, the JavaScript interpreter first computes (or resolves) the expressions to see whether each is true or false; then it computes the entire statement. If an expression resolves to a number other than zero, the expression is considered to be true; if the expression computes to zero, it's considered to be false. Check out the following table for examples of the logical operators available in JavaScript:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parentheses</td>
<td>( )</td>
<td>For calling functions and grouping math expressions</td>
</tr>
<tr>
<td>Unary</td>
<td>(-, ++, --, !)</td>
<td>Decrement, increment, and negation operators</td>
</tr>
<tr>
<td>Mathematical</td>
<td>%, /, *, +</td>
<td>Modulus, division, multiplication, subtraction, addition</td>
</tr>
<tr>
<td>Relational</td>
<td>( \text{&gt;=, &gt;, &lt;=, &lt;} )</td>
<td>Greater than/equal to, greater than, less than/equal to, less than</td>
</tr>
</tbody>
</table>

**Mathematical operators**

Mathematical operators in JavaScript are just as you'd expect: addition, subtraction, multiplication, division, and modulus (the remainder operator). Unlike the assignment operators, which combined assignment with math operations, these operators don't automatically add in the value on the left-hand side of an equation. Take a look at the following examples:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>( x = 1 + 3 )</td>
<td>( x = 4 ) (addition)</td>
</tr>
<tr>
<td>-</td>
<td>( x = 100 - 75 )</td>
<td>( x = 25 ) (subtraction)</td>
</tr>
<tr>
<td>*</td>
<td>( x = 6 \times 7 )</td>
<td>( x = 42 ) (multiplication)</td>
</tr>
<tr>
<td>/</td>
<td>( x = 49 / 7 )</td>
<td>( x = 7 ) (division)</td>
</tr>
<tr>
<td>%</td>
<td>( x = 11 % 5 )</td>
<td>( x = 1 ) (modulus)</td>
</tr>
</tbody>
</table>
Special Operators

A couple of JavaScript operators, `typeof` and `void`, don’t really fit into any other operator category. Take a look for yourself!

**typeof**

You can apply the `typeof` operator to any JavaScript object to find out what type the object is (when you know what type the object is, you then know what you can do to the object). For example, if you apply the `typeof` operator to a string, it returns "string"; if you apply the `typeof` operator to a number, it returns "number"; if you apply the `typeof` operator to true, it returns "boolean" — and so on, for every kind of object that exists. Here’s a glimpse of `typeof` in action:

```javascript
typeof "The Bell Jar"       // returns "string"
typeof true                // returns "boolean"
typeof 69                  // returns "number"
typeof document.lastModified // returns "string"
typeof Math                // returns "function"
typeof someVariable        // returns variable
```

**void**

The `void` operator is a strange beast — it’s used to tell JavaScript to do nothing. Perhaps you think that there’s not much call for an operator that does nothing. Well, you’d be right — except in one pretty important instance: when you want to create and use an object but bypass its requirements for some reason.

For example, suppose that you want to display an image in your Web page. Suppose that you also want to recognize when a user drags her mouse pointer across the image (so that you can display a different message in the status bar for each section of the image the mouse pointer passes over).

Well, you can create an image with the HTML `<img>` tag, but in order to recognize when a user’s cursor passes over that image, you also need to define an additional object, called an `area`.

Now, the reason you need to define an `area` and an `image` is because you want to take advantage of the area’s `onMouseOver` and `onMouseOut` event handlers, but you also want to take advantage of the image’s surface area. How to get the best of both worlds? Define both and tie ‘em together! The only catch is that when you define an area (which is nothing more than a big fat link), it expects you to define a URL to link to. Well, you don’t want to link to anything. All you want to do is use the area’s `onMouseOver` and `onMouseOut` events.

The solution? Give the area what it wants — a URL definition — but give it one that does nothing: in a word, give it `void`! Check out the following code to see exactly how that’s done.

```html
<MAP NAME="thistleMap">
  <AREA NAME="topThistle" COORDS="0.0,228.318"
       HREF="javascript:void(0)"
       onMouseOver="self.status='That mouse pointer sure feels nice'; return true"
       onMouseOut="self.status='Thanks for visiting; come again!'; return true">
</MAP>
```
String operators

Most of JavaScript's operators are designed to work with numeric values. A few, though, are also useful for manipulating strings. In the following table, stringA has been assigned the value "moo" and stringB has been assigned the value "cow".

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition (+)</td>
<td>myString = stringA + stringB</td>
<td>myString = &quot;moocow&quot;</td>
</tr>
<tr>
<td>Append (+=)</td>
<td>myString = &quot;hairy &quot; += stringB</td>
<td>myString = &quot;hairy cow&quot;</td>
</tr>
<tr>
<td>Equality (===)</td>
<td>if (myString == &quot;moocow&quot;)</td>
<td>if myString is equal to &quot;moocow&quot;</td>
</tr>
<tr>
<td>Inequality (!=)</td>
<td>if (myString != &quot;moocow&quot;)</td>
<td>if myString is not equal to &quot;moocow&quot;</td>
</tr>
</tbody>
</table>

Unary operators

Unary operators look a little strange to the uninitiated eye. They're very useful, though, so it's worth spending a minute or two to get familiar with them. In all these examples, x is initially set to 11.

<table>
<thead>
<tr>
<th>Unary</th>
<th>Example (x = 11)</th>
<th>Result/meaning</th>
<th>How come?</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>! (x == 5)</td>
<td>true (negation)</td>
<td>11 isn't equal to 5</td>
</tr>
<tr>
<td>-</td>
<td>x = -x</td>
<td>x = -11 (negation)</td>
<td>Turns positive numbers negative and vice versa</td>
</tr>
<tr>
<td>++</td>
<td>x = ++x</td>
<td>x = 11 (increment by 1)</td>
<td>++ after a var is applied after assignment</td>
</tr>
<tr>
<td></td>
<td>x = ++x</td>
<td>x = 12</td>
<td>++ before a var is applied before assignment</td>
</tr>
<tr>
<td>--</td>
<td>x = --x</td>
<td>x = 11 (decrement by 1)</td>
<td>-- after a var is applied after assignment</td>
</tr>
<tr>
<td></td>
<td>x = --x</td>
<td>x = 10</td>
<td>-- before a var is applied before assignment</td>
</tr>
</tbody>
</table>

Variables

A variable is a named placeholder for a value. You must do three things to a variable (if it is to be of any practical use): declare it, assign a value to it, and access it within its scope. The following sections show you how.

Accessing variables

After you declare a variable, you can then access it. By accessing a variable, I mean you can modify, display, or use the variable's value in a computation. Here's an example:

```javascript
// Variable declaration and assignment
var myTitle = "Princess of the Universe"

// Displaying the value on the screen in a pop-up box
alert("Here is my title: " + myTitle)

// Adding to the value
myTitle += " and everywhere else"

// Comparing one value to another
if (myTitle == "dog catcher") {
    alert("Memo to myself: At least I won the election!")
}
```

Assigning values to a variable

You can assign a value to a variable at the same time you declare it, or at any time after you declare it:

```javascript
// Declaring and assigning all at once
var numberOfWineglasses = 6

// Assigning a value later in the program
numberOfWineglasses = 182

// Assigning a nonsensical value; OK by JavaScript
numberOfWineglasses = "cat food"
```

Declaring variables

Before you can use a variable, you have to declare it. You declare a variable in JavaScript by using the keyword `var`, as shown:

```javascript
var myNumberVariable
var streetAddress
var anArrayOfJobTitles
```
Variables

JavaScript is what’s known as a *loosely typed* language, which means that you don’t have to tell the interpreter what kind of value you’re going to assign to a variable right up front. All that you need is the `var` keyword and a unique variable name of your choice.

The name of your variable must begin with either a letter or an underscore. The variable name can contain numbers but no punctuation marks.

**Understanding variable scope**

A variable is only valid when it’s *in scope*, which means that the variable has been declared between the same curly brace boundaries as the statement that’s trying to access it.

For example, if you define a variable named `firstName` inside a function called `displayReport()`, you can only refer to it inside `displayReport()`’s curly braces. If you try to use the `firstName` variable from inside another function, you get an error. If you want to reuse a variable *among* functions (Eek! A global variable! Quick, call the cops!), you can declare it near the top of your script, before you declare any functions. That way, the variable’s scope is the entire script, from the very first opening curly brace to the last — and all the functions defined within the script get to access it. Take a look at the following code example:

```javascript
function displayReport() {
    var firstName = document.myForm.givenName.value
    alert("Click OK to see the report for " +
           firstName)
    // Using firstName here is fine; it was declared
    // inside the same set of curly braces as the
    // alert() method.
    ...
}
function displayGraph() {
    alert("Here’s the graph for " + firstName) //
    // Error!
    // firstName wasn't defined inside this
    // function's curly braces!
    ...
}
```

As you can see from the comments in the preceding code fragment, it’s perfectly okay to use `firstName` inside the `displayReport()` function; `firstName` is in scope anywhere inside `displayReport()`. It’s *not* okay, however, to use `firstName` inside `displayGraph()`. As far as `displayGraph()` is concerned, no such animal as `firstName` has been declared inside its scope!

---

**HTML Objects: The Heart of It All**

Because JavaScript is an object-based language, much of the JavaScript code you write involves HTML objects. Part II shows the relationship of each object in the JavaScript object hierarchy to one another, which provides a good overview of all the objects available to you. This part, though, is where you find all the details about the HTML objects used in JavaScript, the properties and methods that each object contains, and the event handlers that each object supports.

The sections in this part are cross-referenced so that you can look things up in different ways. For example, if you want to find out about a particular object, you can find it alphabetized in this part. On the other hand, if you know the name of a method but can’t remember what objects are associated with it, check out Part VI, “Methods: How an Object Behaves.” Part VI refers you to the objects each particular method supports.

**In this part . . .**

- Creating HTML objects (reprise)
- Invoking an object's methods
- Accessing an object's properties
About Objects

The two-bit definition of an object is “a software representation of some useful thing.” JavaScript objects are no exception: Each of them are representations of the things that you need to build Web pages — push buttons, input fields, dates, and so on. Part I explains how to create each of the HTML objects in detail. This section focuses on the way these objects can be manipulated (that is, how their properties can be accessed and their methods invoked) with JavaScript statements.

For each element, you find three separate sections:

♦ How to create the object
♦ How to access the object’s properties
♦ How to invoke the object’s methods

For more detailed HTML syntax on how to create each object listed, see also Part I.

Remember that you create objects in HTML, but you work with objects in JavaScript. Therefore, you should place the code fragments in “Creating an object” between <HTML>...</HTML> tags. In contrast, you should place the code fragments in the “Accessing an object’s properties” and the “Invoking an object’s methods” sections between <SCRIPT>...</SCRIPT> tags.

anchor

An anchor is a piece of text that uniquely identifies a spot on a Web page. After you define an anchor — say, in the middle of a page — you (or anyone else for that matter) can set up a link so that when a user clicks the link, the page loads right where the anchor is located.

Creating an anchor:

<A NAME="TOC">Table of Contents</A>

Accessing anchor properties:

document.anchors.length

Invoking anchor methods:

The anchor object has no associated methods.

applet

The applet object corresponds to a Java applet embedded in an HTML form.

Creating an embedded applet:

<APPLET NAME="NervousApplet"
CODE="NervousText.class" width=400 height=50>
<PARAM NAME="text" VALUE="Enter your text here.">
</APPLET>

Accessing applet properties:

The properties available to you depend on the specific Java applet with which you’re working. One property should be available for all Java applets, however, and that’s the name property:

document.applets[0].name

Invoking applet methods:

The applet methods available to you depend on the specific Java applet with which you’re working. Ask the person who developed the Java applet that you’re including in your Web page for a list of public methods that you can invoke on the applet.

area

The area object is used to make a specific area of an embedded image responsive to user events. You can make an area respond to a click event or to mouse pointer movement events.

Three separate HTML entities need to be defined as part of an area: an area (of course), an HTML <MAP>, and an image. A complete example awaits you in the following code.

Creating an area:

<MAP NAME="thistleMap">

<AREA NAME="topThistle" COORDS="0,0,228,318"
HREF="javascript:displayMessage()"
onMouseOver="self.status='When you see this message, click your left mouse button'; return true"
onMouseOut="self.status=''; return true">

</MAP>

<IMG NAME="currentImage" SRC="images/thistle.gif" ALIGN="MIDDLE"
ALT="[Scottish thistles]" USEMAP="#thistleMap"
Instead of defining a URL value for the \texttt{HREF} attribute, the line looks like this:

\texttt{HREF="javascript:displayMessage()"}

This statement tells the JavaScript interpreter that it should invoke the custom function \texttt{displayMessage()} when a user clicks on this area, \textit{not} load a URL.

If you want to use the area event handlers \texttt{onMouseOver} and \texttt{onMouseOut}, but you don't want anything to happen when the user \textit{clicks} on the area, assign the \texttt{HREF} attribute of \texttt{<AREA>} equal to this: \texttt{"javascript:void(0)"}.

\textbf{Accessing area properties:}

No, this isn't a mistake — areas are stored in the \texttt{links} array.

\begin{verbatim}
document.myForm.inStateButton

document.myForm.inStateButton.name
document.myForm.inStateButton.type
document.myForm.inStateButton.value
\end{verbatim}

A button object creates a push button on an HTML form.

\begin{verbatim}
\texttt{<FORM NAME="myForm">}
\texttt{<INPUT TYPE="button" NAME="inStateButton" VALUE="In State" onClick="display(this)">}
\end{verbatim}

In this example, the name of the button is \texttt{inStateButton}, and the text displayed on the face of the button is "In State". When a user clicks on this button, the JavaScript interpreter calls the \texttt{display()} function and passes it one argument: the entire \texttt{inStateButton} object (denoted by this).

\textbf{Accessing button properties:}

\begin{verbatim}
document.myForm.inStateButton.name
document.myForm.inStateButton.type
document.myForm.inStateButton.value
\end{verbatim}

The preceding JavaScript code fragments represent the button's name (\texttt{inStateButton}), type ("button"), and value ("In State"), respectively.

\textbf{Invoking button methods:}

\begin{verbatim}
document.myForm.inStateButton.click()
\end{verbatim}

\texttt{// used to click the button programmatically}

\textbf{checkbox}

A checkbox object creates — well, a check box. A check box is like a toggle switch — its value is always either off or on.

\begin{verbatim}
\texttt{<INPUT TYPE="checkbox" NAME="jazzCheckbox" CHECKED value="jazz" onClick="display(this)">}
\end{verbatim}

Click here if you like jazz.

In this example, the name of the checkbox is \texttt{jazzCheckbox}. The value is "checkedJazz" (which is the value that is submitted to the CGI program if the check box is checked). The checkbox is automatically checked the first time that it appears to the user; the text that appears next to the check box is "Click here if you like jazz."

\textbf{Accessing check box properties:}

\begin{verbatim}
document.myForm.jazzCheckbox.checked
document.myForm.jazzCheckbox.defaultChecked
document.myForm.jazzCheckbox.name
document.myForm.jazzCheckbox.type
document.myForm.jazzCheckbox.value
\end{verbatim}

\textbf{Invoking check box methods:}

\begin{verbatim}
document.myForm.jazzCheckbox.click()
\end{verbatim}

\texttt{// used to set checkbox programmatically}

\textbf{document}

A document object defines characteristics of the overall body of a Web page, such as the background color of a page, the default text color, and so on.

\begin{verbatim}
\texttt{<BODY BGCOLOR="lime" TEXT="maroon" LINK="red" VLINK="blue"}
\texttt{onLoad="welcome()"}
\texttt{onUnload="goodbye()">}\</BODY>
In Netscape Navigator 3.0, the fileUpload object is read-only (you can't change its properties).

A form is used to gather input from users and to post data (including user input) to a server for additional processing.

Creating a form:

```html
<FORM NAME="myForm" METHOD="POST" TARGET="_parent"
    ACTION="http://altavista.digital.com/cgi-bin/query?q=what=web&fmt=.q=JavaScript"
    ENCTYPE="multipart/form-data"
    onSubmit="return verifyComplete()">
    </FORM>
```

Accessing form properties:

- `document.myForm.action`
- `document.myForm.elements[0].name`
- `document.myForm.elements[0].value`
- `document.myForm.encoding`
- `document.myForm.length`
- `document.myForm.method`
- `document.myForm.target`
- `document.forms[0]` // first form defined in the document
- `document.forms.length` // total # forms defined

Invoking form methods:

- `document.myForm.reset()`
- `document.myForm.submit()`

A frame is a special kind of window. You can think of a frame as an individual pane of glass — that is, you can have several frames per regular window or just one. A user can scroll each frame independently.

Creating a frame:

```html
<FRAMESET ROWS="50%,50%" COLS="40%,60%">
    <FRAME SRC="framcon1.html" NAME="frame1">
    <FRAME SRC="framcon2.html" NAME="frame2">
</FRAMESET>
```

Accessing frame properties:

- `frames.length`
- `frames[0].name`
hidden

A hidden element is an input text field that doesn't appear on-screen.

Creating a hidden element:

```html
<INPUT TYPE="hidden" NAME="markupPercent" VALUE=80>
```

Pretty straightforward, isn't it? The preceding line of code defines a hidden element named markupPercent and stuffs it with the initial value of 80 so that subsequent calculations can add an 80 percent markup to any purchase. (I think you can guess why a programmer might want to hide a value like this!)

Accessing hidden properties:

```javascript
document.myForm.markupPercent.name
document.myForm.markupPercent.type
document.myForm.markupPercent.value
```

Invoking hidden methods:

The hidden object has no associated methods.

history

The history object contains a linked list of all the URLs that a user has visited from within a particular window. This object provides the list of URLs you see when you select the Go menu item in Navigator or Internet Explorer.

Creating a history object:

This one's a freebie; you don't have to write a lick of code to define the history object! The history object is defined for you by your Web browser. Each time you load a new Web page, your browser automatically adds that URL to the history object.

Accessing history properties:

```javascript
history.length
```

Invoking history methods:

```javascript
history.back()
history.forward()
history.go(-3)
```

image

The image object in JavaScript corresponds to the HTML `<img>` element. The image object represents an image embedded into a Web document.

At the time of this writing, Internet Explorer 3.0 does not support the image object.

JavaScript provides an image constructor (the `new` keyword described in Part II) that you can use to create and load an image behind the scenes. You may want to load an image behind the scenes if you have a large image that you want to load while your user is doing something else. That way, when it's time to display the image, the image will already be loaded into memory so it will display right away. Here's how it works:

// To load the image, do this:
// first, create an image variable
var myImage = new Image();

// then, load the image into memory
myImage.src = "images/thistle.gif"

// Finally, to display the image, do this:
document.images[0].src = myImage.src

Creating an image:

```html
<IMG NAME="currentImage"
SRC="images/jazzman.gif" ALIGN="MIDDLE" ALT="[Jazz guitarist]"
USEMAP="#jazzMap"/>
```

Accessing image properties:

```javascript
document.myForm.currentImage.border
document.myForm.currentImage.complete
document.myForm.currentImage.complete
document.myForm.currentImage.hspace
document.myForm.currentImage.isspace
document.myForm.currentImage.name
document.myForm.currentImage.prototypeName
document.myForm.currentImage.src
document.myForm.currentImage.vspace
document.myForm.currentImage.width
```

Invoking image methods:

No methods are associated with the image object.
link

A link is a piece of text (or an image) that loads another Web page when a user clicks on it. (A link often loads a specific spot, or anchor, on another Web page.)

Creating a link:

\[
\text{<A HREF="#TOC" onClick="verifyData()" onMouseOver="displayScrollingText()" >Back to Table of Contents</A>}
\]

Accessing link properties:

- `document.links.length`
- `document.links[0].hash`
- `document.links[0].host`
- `document.links[0].hostname`
- `document.links[0].href`
- `document.links[0].pathname`
- `document.links[0].port`
- `document.links[0].protocol`
- `document.links[0].query`
- `document.links[0].target`

Invoking link methods:

Neither the link object nor the links array has any associated methods. (For more about arrays, see also Part IV.)

location

Think of the location object as a mini-version of the history object. Instead of holding information on all the recently visited URLs, like the history object does, the location object contains information about just one URL — the one that's currently loaded.

Creating a location object:

Just like the history object, you don't have to write any code to define the location object — it's set up for you automatically by your Web browser.

Accessing location properties:

- `location.hash`
- `location.host`
- `location.hostname`
- `location.href`
- `location.pathname`
- `location.port`
- `location.protocol`
- `location.search`
- `location.target`

Invoking location methods:

The location object has no associated methods.

Math

The Math object is the only built-in object in JavaScript. Unlike a built-in data type (such as the ones shown in Part IV, which all describe a class of thing), the Math object not only describes a class of thing all right (a mathematical system) — but it also implements the class. Math contains properties and methods for all kinds of mathematical constants and functions, such as logarithms and square roots. Hey, why reinvent the wheel?

Creating a Math object:

You never define the Math object yourself; it's done for you automatically. All you have to do is reference it, as shown in the following examples.

Accessing Math properties:

- `Math.E`
- `Math.LN2`
- `Math.LN10`
- `Math.LOG2E`
- `Math.LOG10E`
- `Math.PI`
- `Math.SQRT1_2`
- `Math.SQRT2`

Invoking Math methods:

- `Math.abs(23)`
- `Math.acos(123)`
- `Math.asin(1)`
- `Math.atan(1)`
- `Math.atan2(1)`
- `Math.ceil(25.85)`
- `Math.cos(0)`
- `Math.exp(1)`
- `Math.floor(25.85)`
- `Math.log(1)`
- `Math.max(5,15)`
- `Math.min(5,15)`
- `Math.pow(8,3)`
- `Math.random()`
- `Math.round(15.58)`
- `Math.sin(0)`
- `Math.sqrt(49)`
- `Math.tan(0)`
navigator

The navigator object contains information about the version of Navigator currently in use.

Creating a navigator object:

You never define the navigator object yourself; it's done for you automatically when you bring up your Web browser.

Accessing navigator properties:

navigator.appCodeName
navigator.appName
navigator.appVersion
navigator.complete

// The mimeTypes array elements must be accessed
// through the array, as shown
navigator.mimeTypes[0].description
navigator.mimeTypes[0].enabledPlugin
navigator.mimeTypes[0].suffixes
navigator.mimeTypes[0].type

// The plugins array elements must be accessed
// through the array, as shown
navigator.plugins[0].filename
navigator.plugins[0].length
navigator.plugins[0].name

navigator.userAgent

Invoking navigator methods:

navigator.javaEnabled()

password

A password object is a special text input field that displays asterisks on-screen in place of the characters that the user actually types, enabling users to type in sensitive information (like a password or financial information) without fear that someone peeking over their shoulder will get a glimpse.

Creating a password:

<INPUT TYPE="password" NAME="userPassword" SIZE=15 VALUE="secret!">

Accessing password properties:

document.myForm.userPassword.defaultValue
document.myForm.userPassword.name
document.myForm.userPassword.type
document.myForm.userPassword.value

radio

A radio button is a toggle switch, something like a check box.

Unlike a check box, though, radio buttons are often grouped in sets to allow users to select a single option from a list.

Creating a radio button:

What's your favorite time of day?

<INPUT TYPE="radio" NAME="timeChoice" VALUE="morningSelected" CHECKED onClick="showValues(0)"> Morning
<INPUT TYPE="radio" NAME="timeChoice" VALUE="afternoonSelected"
onClick="showValues(1)"> Afternoon
<INPUT TYPE="radio" NAME="timeChoice" VALUE="eveningSelected"
onClick="showValues(2)"> Evening

Accessing radio properties:

document.myForm.timeChoice[index].checked
document.myForm.timeChoice[index].defaultChecked
document.myForm.timeChoice[index].length // # of radio buttons

Invoking password methods:

document.myForm.userPassword.focus()
document.myForm.userPassword.blur()
document.myForm.userPassword.select()
reset

A reset object is a special kind of button. When a user clicks on a reset button, it clears out all the user-input values in a form and resets each field to its default value.

Creating a reset button:

```
<INPUT TYPE="reset" NAME="resetButton" VALUE="Reset"
       onClick="reinitializeFormulas()"/>
```

Accessing reset properties:

document.myForm.resetButton.name
document.myForm.resetButton.type
document.myForm.resetButton.value

Invoking reset methods:

document.myForm.resetButton.click()

select

The select object is used to display both a single-selection list box and a scrolling multiple-selection list box.

Creating a select element:

```
<SELECT NAME="fave" SIZE=3 MULTIPLE
       onBlur="displayResult(this)"
       onChange="validate()"
>
  <OPTION VALUE="popChosen" SELECTED> pop
  <OPTION VALUE="rockChosen" > rock
  <OPTION VALUE="dogsChosen" > dogs barking Jingle Bells
  <OPTION VALUE="rapChosen" > rap
  <OPTION VALUE="showChosen" > show tunes
  <OPTION VALUE="africanChosen" > African classical
</SELECT>
```

Accessing select properties:

document.myForm.fave.length
document.myForm.fave.name
document.myForm.fave.options
document.myForm.fave.selectedIndex
document.myForm.fave.options[index].defaultSelected

Invoking select methods:

document.myForm.fave.blur()
document.myForm.fave.focus()

Submit

A Submit object is a special kind of button that submits all of the user input values on a form to the server when a user clicks on the button.

Creating a Submit button:

```
<INPUT TYPE="submit" NAME="submitButton"
       VALUE="Submit Form" onClick="verifyInput()"
>
```

Accessing submit properties:

document.myForm.submitButton.name
document.myForm.submitButton.type
document.myForm.submitButton.value

Invoking submit methods:

document.myForm.submitButton.click()

Clicking on the Submit button sends a form to the URL value specified in the form's HTML ACTION attribute. The data is sent as a series of attribute-value pairs, each pair separated by an ampersand (&).

text

The text object is a single-line input field. (If you want a multiple-line input field, see the textarea object, defined in the following section).

Creating a text element:

```
<INPUT TYPE="text" NAME="lastName" VALUE="your name here"
       SIZE=30 onBlur="validate()" onChange="validate()"
>
```

Accessing text properties:

document.myForm.lastName.defaultValue
document.myForm.lastName.name
document.myForm.lastName.type
document.myForm.lastName.value
Invoking text methods:

doctype.myForm.lastName.focus()
doctype.myForm.lastName.blur()
doctype.myForm.lastName.select()

textarea

A textarea object is just like a text object, except that instead of defining one scrolling input line, the textarea object defines a multi-line scrolling text box.

Creating a textarea object:

<textarea NAME="directions" ROWS=4 COLS=60 onBlur="validate()" onChange="display()"
onFocus="welcome()" onSelect="changeMode()">
This is default text. You can type right over it, add to it, cut it, paste it, or copy it.
</textarea>

Accessing textarea properties:

doctype.myForm.directions.defaultValue
doctype.myForm.directions.name
doctype.myForm.directions.type
doctype.myForm.directions.value

Invoking textarea methods:

doctype.myForm.directions.focus()
doctype.myForm.directions.blur()
doctype.myForm.directions.select()

window

The window object is the top-level granddaddy object for all document objects, and you're given the first one gratis, compliments of the <BODY>...</BODY> tag pair. If you want to create extra windows, the following example is the way to do it.

If you do create multiple windows, remember to tack the new window's name (myOtherWindow, for example) in front of any associated property or method you access so that the interpreter knows which window you're referring to at all times.

Creating a window:

Here's the generic syntax for creating a window, followed by an example:

windowVar = open("URL", "windowName", [
  "windowFeatures"])

myOtherWindow = open("win2.html", "secondWindow", 
  "toolbar=yes, location=yes, directories=yes, 
  status=yes, menubar=yes, scrollbars=yes, 
  resizable=yes, width=250, height=400")

Accessing window properties:

defaultStatus
frames
length
name
opener
parent
self
status
top
window

Invoking window methods:

alert("Form will be sent now...")
blur()
close()
confirm("Do you really want to quit?")
focus()
myOtherWindow = open("", "secondWindow", 
  "toolbar=yes, location=yes, directories=yes, 
  status=yes, menubar=yes, scrollbars=yes, 
  resizable=yes, width=250, height=400")
prompt("Enter the file name:", "testfile.txt")
scroll(50, 100)
timeoutID=setTimeout("displayAlert()", 2500)
clearTimeout(timeoutID)
myOtherWindow.write("Here is myOtherWindow.")
Data Types: Building Basic JavaScript Objects

Some kinds of objects, such as dates and strings, are really basic; they're so basic, in fact, that almost every script you'll ever write will probably contain a reference to them. Because they're so common, JavaScript thoughtfully provides built-in classes for them — but because JavaScript is loosely based on C, these classes are still referred to by the conventional name of data types.

Unlike the objects that you find in Part III, JavaScript data types don't have corresponding HTML tags — they're pure JavaScript constructs. You declare them inside the `<SCRIPT>`...`</SCRIPT>` tag pair by using JavaScript statements, as you can see in the following sections.

In this part . . .

- Creating objects based on built-in data types
- Accessing object properties
- Invoking object methods
Array

An Array is an indexed list of things called elements. Element values can be whatever you want them to be — numbers, strings, or even other objects. You can fill an array with elements when you create it by passing values to the array constructor, or you can create an empty array and fill it with elements later.

Arrays are useful whenever you want to keep track of a group of related items.

Syntax:

```
arrayName = new Array([element1, element2, ...
   elementN | arraySize])
```

Example:

```
var listOfPets = new Array("dog", "cat", "gerbil")
listOfPets[3]="bird"
// signals that 2 initial elements are expected,
// but
// the interpreter won't complain if you assign
// more
var favoriteFoods = new Array(2)
favoriteFoods[0]="frozen yogurt"
favoriteFoods[1]="barbecued beef"
favoriteFoods[2]="stir-fry" // 3 elements is okay
var toDoList = new Array()
```

Accessing array properties:

```
favoriteFoods.length
favoriteFoods.prototypeName
```

Invoking array methods:

```
favoriteFoods.join()
favoriteFoods.reverse()
favoriteFoods.sort()
```

The first two lines of the preceding example code create an array called listOfPets, which contains four entries: the first entry is “dog”, the second, “cat”, and so on. Here’s a conceptual look at the listOfPets array:

```
[0] “dog” // first element in the listOfPets array is “dog”
[1] “cat” // second element in the listOfPets array is “cat”
```

Date

A Date object in JavaScript is just what it is in real life — a specific time including second, minute, hour, day, month, and year information. Any time you work with dates, you should use the Date object. You can create dates based on the current time or on values that you provide. You can modify and manipulate the date then to your heart’s content.

Syntax:

```
dateName = new Date()
// if no parameters are passed to the constructor,
// the result is the current date/time

dateName = new Date("month day,
   year hours:minutes:seconds")

dateName = new Date(year, month, day)
dateName = new Date(year, month, day, hours,
   minutes, seconds)
```

Example:

```
var today = new Date()
var birthday = new Date("October 21, 1973
   01:40:00")
var graduation = new Date(1990, 8, 6)
var wedding = new Date(92, 07, 12, 10, 30, 21)
```

Accessing Date properties:

You access the contents of a Date a little differently than you do the contents of other data types. Accessing Date properties (except one) is done by calling methods, as you can see in the following example.

One property, called prototype, is different from all the rest. You can use a prototype to create a new property for a Date object, and you can name the new property you create anything you want. For example, every instance of Date has built-in properties like hours, minutes, seconds, day, month, and year. Using prototype, you can add a brand-new property that’s meaningful to you to all instances of Date. Check out the following example to see how to add a property called description to a date.
// create a new date variable called "today"
var today = new Date();

// create a new property for date objects called "description"
Date.prototype.description = null

// assign a value to the new date object property
today.description = "Today I lost my first tooth!

In the preceding code, you first add the description property to the Date object and then add a value for the description property. For example, "Today I lost my first tooth!" is defined for the instance of Date called today.

**Invoking Date methods**

If you think about it, you can basically do only two things with a property after it's defined: You can get the value of the property to see what it is, and you can set the value of the property. Data types have predefined properties; you never have to define them yourself. All you have to worry about is getting values (which, for instances of Date, you do with what's affectionately known as *getter* methods, or *getters* for short) and setting values (using *setter* methods, or *setters*).

**Date getters:**

```javascript
birthday.getDate()
birthday.getDay()
birthday.getHours()
birthday.getMinutes()
birthday.getMonth() // subtracts 1 from the correct value
birthday.getSeconds()
birthday.getTime()
birthday.getTimeZoneOffset() // undefined in Navigator 2.0
birthday.getFullYear()
birthday.toGMTString()
birthday.toLocaleString()
Date.UTC(1983, 6, 24, 2, 51, 8)
```

**Date setters:**

```javascript
birthday.setTime(Date.parse("January 7, 1997"))
birthday.setDate(21)
birthday.setHours(3)
birthday.setMinutes(59)
birthday.setMonth(11)
birthday.setSeconds(50)
birthday.setTime(identicalTwinsBirthday.getTime())
birthday.setYear(82)
```

**Function**

In Part V, you see how you normally declare functions. In Navigator 3.0, you can use an alternative way to create a function—you can use the built-in Function object.

**Syntax:**

```javascript
functionName = new Function([arg1, arg2, ... argN],
                           FunctionBody)
```

**Example:**

```javascript
var calcDogYears = new Function("age", "return age * 7")
alert("If Spot is 4, he's \" + calcDogYears(4) + \" in people years")
```

The preceding example code defines a variable called calcDogYears, of type Function, which takes one parameter (age), and returns a value (specifically, whatever the value of age is, multiplied by seven). Then the function is called like this: calcDogYears(4)

When you declare a function, JavaScript compiles it once. On the other hand, when you create a function by using the Function object, the JavaScript interpreter has to evaluate the function each time it's called. You gain flexibility with the Function object because you can create new functions on the fly (perhaps based on some user input). However, you pay a price in reduced efficiency by using the Function object—your script may take longer to execute.

**Accessing Function properties:**

```javascript
calcDogYears.arguments[0]
calcDogYears.arguments.length
calcDogYears.prototypeName
```

**Invoking Function methods**

The Function object has no associated methods.

**Option**

You can use an Option object to represent one of the options assigned to a single- or multiple-selection list box (a select element). You create the Option object first and then insert it into the associated select element's options array.
The Option constructor accepts four optional arguments, as shown in the following table:

<table>
<thead>
<tr>
<th>Optional Arguments for the Option Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optionText</td>
<td>The text to display in the select list next to this option</td>
</tr>
<tr>
<td>optionValue</td>
<td>The value to be sent to the server when this option is selected</td>
</tr>
<tr>
<td>defaultSelected</td>
<td>Whether this option should be selected by default (true or false)</td>
</tr>
<tr>
<td>selected</td>
<td>Whether this option is currently selected (true or false)</td>
</tr>
</tbody>
</table>

**Syntax:**

```javascript
optionName = new Option([optionText, optionValue, defaultSelected, selected]);
```

```javascript
selectName.options[index] = optionName
```

**Example:**

```javascript
var blues = new Option("Blues music", "blues", true, true);
document.myForm.musicSelection[0] = blues
```

You can delete an Option from the options array of a select element like this:

```javascript
document.myForm.musicSelection[0] = null
```

Deleting an element in an array causes the array to be compressed together, so if you delete the second element of an array containing three elements, the array then holds two elements — the first and the third.

**Accessing Option properties:**

- blues.defaultSelected
- blues.index
- blues.prototypeName
- blues.selected
- blues.text
- blues.value

**Invoking Option methods**

No methods are associated with the Option object.

---

A String object is neither more nor less than a series of characters, usually surrounded by quotes, like this: "Ralph", "Henrietta and Bugsy", "123,456,789.00", or "1600 Pennsylvania Avenue". Using strings is the only way you can pass pieces of text around inside JavaScript. Unless you expect to do some arithmetic operations on a value, you probably want to work with the value in string form, which you can see exactly how to do in this section.

**Syntax:**

Two ways exist to create a string. One way is to use the built-in String data type; the other way is simply to surround the string value with double quotes ("like this"). Strings can be stored in variables but they don't have to be; when strings are not stored in variables, they're called string literals.

```javascript
stringName = new String("some string"); // one way
document.myForm.musicSelection[0] = stringName
```

```javascript
stringName = "some string" // another way
```

**Example:**

```javascript
var lastName = "Smith" // a string variable
var firstName = new String("Barney") // a string variable
alert("Millions, including the " + lastName + ", enjoy JavaScript daily with their morning pasty.");
```

**Accessing String properties:**

- lastName.length
- String.prototype.description = null
- middleName = new String("T.J.")
- middleName.description = "T.J. doesn't actually stand for anything."

**Invoking String methods:**

- lastName.anchor("tableOfContentsAnchor")
- lastName.big()
- lastName.blink()
- lastName.bold()
- lastName.charAt(3)
- lastName.charCodeAt()
- lastName.fontcolor("springgreen")
- lastName.fontsize(7)
- lastName.indexOf("i")

(continued)
Functions

A function is nothing more than a named group of JavaScript statements that, when called, execute all at once. Unlike methods, which are always associated with a particular object's data, functions don't belong to any particular object. Instead, functions adopt the values you pass to them (called arguments) when you invoke the function.

In this part . . .

- Understanding functions
- Creating your own custom functions
- Using functions to create instances of objects
- Calling built-in functions
About Using Functions

A function is nothing more than a named group of JavaScript statements. Every time you use, or call, the function, all the statements in the function are executed in one fell swoop. Although JavaScript provides a handful of built-in functions, you're not nearly as likely to use them as you are to use the custom functions that you define. You can find a complete list of all the built-in JavaScript functions, as well as steps for how to create your own functions, later in this part. Whether they're built-in or custom-designed, though, you call all functions (with the exception or construction functions) the very same way.

Calling a function

Remember, you must first define a function before you can call it.

Suppose you have access to a function called calculateTotal() (maybe you borrowed it, maybe you wrote it, maybe you purchased it). The purpose of this function is to calculate a total price based on the number of items a user wants to order and the price of each item.

Not surprisingly, this function takes two input parameters: one for the price, and the other to represent the number of items.

calculateTotal() returns a number.

When you want to call a function, it's essential that you know three things:
- The correctly spelled name of the function
- The number and type of parameters the function expects
- What the function is supposed to return; for example, a number, a string, or whatever

The best way to find out the answer to these questions is to look at the function definition. If you created the file, the definition should be somewhere near the top of the HTML file. (Of course, if the function you're interested in calling is one of the built-in ones, it won't be defined in the HTML file — it's built right in! See also "Built-in Functions" later in this part for details on calling built-in functions.)

Since you know the name of the function (it's calculateTotal()), what it accepts (two numbers, one for the number of items and one for the price), and what it returns (a number), you can call the function, as shown in the following code:

```javascript
var myPrice = calculateTotal(3, 19.95)
```

Defining a function

Creating, or defining, a function is easy as pie (whoever coined that phrase must have been better at rolling out pie crust than I am!). Here's the generic syntax for a function declaration:

```javascript
function aName([parameter][, parameter]
    [,..., parameter]){
    statements
}
```

Here's how the calculateTotal() function that was called in the preceding section may have been defined:

```javascript
function calculateTotal(numberOrdered, itemPrice){
    var result = numberOrdered * itemPrice
    return result
}
```

As you'd expect, this function multiplies the value for numberOrdered by the itemPrice and returns the result. See also "Creating a Custom JavaScript Function" later in this part.

Returning values

The return keyword is used to hand a value from a function back to whatever line of code called the function in the first place. The calling line of code can then use the returned value for anything it wants (to display it, use it in further calculations — that kind of thing). Technically, a function doesn't have to return a value but, in practice, most of them do. Here's the syntax for return:

```javascript
return expression
```

You can see by the syntax that a function can return an expression, and an expression can be just about anything: a variable, a statement, or a complex expression. Check out these examples:

```javascript
// returning a variable
return calculatedResult

// returning a statement
return (inputValue * 10)

// returning a complex expression
return (someValue / 100 + ((anotherValue * 55) % 9))
```

Make sure that the return statement is the very last statement in the body of your function. After all, return means just that — return. When the JavaScript interpreter hits the return statement, it returns to whatever line of code called it, right then and there, and continues interpreting the script. If you've placed statements inside the function after the return statement, they'll never be evaluated.
Built-in Functions

"Creating a Custom JavaScript Function," later in this part, describes how to create your very own functions. The handful of functions in this section, though, are freebies — they've already been created and are ready and waiting for you should you ever want to call them. Read on for real-life examples.

**escape() — encodes a string**

Use the `escape()` function to encode special characters such as spaces, tabs, dollar signs, hash marks, exclamation points, and so on — so that the characters can be sent safely from one program (your Web page) to another program (a CGI program on a server). Then you can use `unescape()` on the other side, inside the CGI program, to decode the characters.

The string that returns from `escape()` is in the form "%xx", where xx is the ASCII encoding of each character in the argument. The `string` argument must be a nonalphanumeric string in the ISO Latin-1 character set (which, translated into English, means any character that's not an alphabetic character or a number). If you pass `escape()` a string containing numbers and punctuation marks by mistake, like this:

```javascript
escape("67%@x")
```

`escape()` doesn't even attempt to encode the string — the function just returns the same string that you sent it.

**Syntax:**

`escape(string)`

**Example:**

```javascript
encodedStringToPass = escape("& ")
// The above statement will return "%26%20"
```

**eval() — evaluates a string**

The `eval()` function evaluates a string that contains a JavaScript phrase (as opposed to a regular old JavaScript phrase that's not inside a string) and returns the value. Sometimes the application of this function is really straightforward: Someone enters a number into a text field (for example, "82345") and you need to turn the value into an integer before you can use it in any numeric computations. (See also "isNaN()" in the following section.) Other times, the application can be a bit more sophisticated, as shown in the following example.

```javascript
Syntax:

eval(string)

Example:

```javascript
var totalPrice = "((numberOrdered * price) * tax)"
if (noTaxRequired) {
  totalPrice = "(numberOrdered * price)"
}

// totalPrice prints out as a numeric value,
// not the string you see above
document.write(eval(totalPrice))
```

The preceding code snippet evaluates `totalPrice` (which has a string value) and displays the result on-screen.

**isNaN() — is not a number**

The `isNaN()` function lets you determine whether a value is Not a Number. (Get it — `isNaN()?). This function returns `true`, or `1`, if the specified `testValue` is not a number. The function returns `false`, or `0`, if the `testValue` is a number. Your fifth-grade English teacher was right — double negatives are confusing! Unfortunately, many programming languages make liberal use of them.

**Syntax:**

`isNaN(testValue)`

**Example:**

```javascript
if (isNaN(ageInputValue)) {
  alert("Please enter a number for the age field.")
}
```

**parseFloat() — turns strings into floating-point numbers**

A floating-point number is any number that has a decimal in it (19.95, for example). Surprisingly, it's handy at times to be able to turn a string into a floating-point number. For example, what if you have the value "78.95" in one of your input fields and you want to use it in some calculations? As it stands, you can; it's a string (you know it's a string because it's got quotes around it), and you can't do much in the way of mathematical calculations on strings. The solution is to convert the string value to a floating-point value by using the `parseFloat()` function and then perform your calculations.
If the string argument that you give `parseFloat()` can’t be converted completely, the `parseFloat()` function behaves in one of two ways:

- If the very first character can’t be converted, `parseFloat()` returns "NaN" (Not a Number).
- If the first character can be converted but a subsequent character can’t, `parseFloat()` returns the floating-point value of everything it could convert, up until it encountered the invalid character.

Valid characters for the string argument include the numbers 0 through 9, plus (+), minus (−), decimal (.), and exponent (E).

**Syntax:**

`parseFloat(string)`

**Example:**

```javascript
function isANumber(inputValue){
    answer=true
    for (var i=0; i<inputValue.length; i++) {
        if ($(inputValue.charAt(i) != "0") && !parseFloat(inputValue.charAt(i))) {
            answer=false
            break
        }
    }
    return answer
}
```

The function in this example returns false if the argument sent to it is not a number, and true if it is a number. All the action is happening inside the if statement (which is itself buried inside a for loop). The if statement looks through the input value one character at a time and stops the minute that it encounters a character that can’t be represented as a number.

**`parseInt()` — turns strings into integers**

Use the `parseInt()` function to turn specified string and radix arguments into an integer.

A radix isn’t a strong-tasting root; it’s a representation of a numbering system. The radix argument tells `parseInt()` which base you want the string converted to — for example, decimal (radix = 10), octal (radix = 8), hexadecimal (radix = 16), binary (radix = 2), and so on. Fortunately for the nongeeks among us, decimal is the default. If you don’t specify a radix, the `parseInt()` function returns an integer in good old base 10, the very same numbering system human beings use to communicate. (The `parseInt()` function does convert strings to integers of different bases, depending on what type of character is first in the string.)

If the string argument that you give `parseInt()` can’t be converted completely, the function behaves in one of two ways:

- If the very first character can’t be converted, `parseInt()` returns "NaN" (Not a Number).
- If the first character can be converted but a subsequent character can’t, `parseInt()` returns the integer value of everything that it could convert up until it encountered the invalid character.

Valid characters for the string argument include the numbers 0 through 9 plus any other characters that are allowed by the radix you choose.

**Syntax:**

`parseInt(string [ , radix])`

**Example:**

```javascript
parseInt("F", 16) // base 16; returns 15
parseInt("17", 8) // base 8; returns 15
parseInt("15", 10) // base 10; returns 15
parseInt("15.99", 10) // base 10; returns 15
parseInt("123") // base 10; returns 123
parseInt("11111", 2) // base 2; returns 15
```

**unescape() — decodes a string**

As you can probably guess, `unescape()` is the opposite of `escape()`. The `escape()` function encodes (or escapes) a string, and `unescape()` decodes (or unescapes) a string. The string argument can be in either of two forms:

- %xx, where xx is an integer between 0 and 255
- A hexadecimal number between 0x0 and 0xFF

The returned string will be a series of characters in the ISO Latin-1 character set (better known as the "non-numeric, non-punctuation-mark character set").

**Syntax:**

`unescape("string")`

**Example:**

```javascript
decodedString = unescape("%26%20") // returns "& "
```
Creating a Custom JavaScript Function

To create, or declare, a function in JavaScript, you need to use the function keyword as shown in the following example:

```javascript
function functionName([parameter] [, parameter] [
    statements
    return returnExpression
]
)
```

As you can see in the preceding syntax, functions can accept parameters (also called arguments) — but they don't have to (the square brackets, as always, denote optional items). Here's the declaration for a function that accepts no parameters:

```javascript
function displayCopyright() {
    alert("The content in this Web page is copyrighted by the XYZ Corporation.");
    return true
}
```

And here's how you might call it:

```html
<input type="button" name="displayCopyButton" value="View Copyright Info"
    onclick="displayCopyright()"/>
```

With a fixed number of parameters

Here's an example of a function named computeTax(), that requires two parameters: price and taxRate.

```javascript
function computeTax(price, taxRate) {
    return (price * taxRate)
}
```

```html
<input type="button" name="myButton" value="Press me"
    onclick="computeTax("19.95", .08)"/>
```

With a variable number of parameters

You don't have to do anything special to declare a JavaScript function that accepts a variable number of parameters. All you need to do is send however many parameters you want to send to a regular function. Then, in the body of the function, you can capture all of the parameters and process them as you see fit. For example:

```javascript
function myFunction(inputValue) {
    for (var i=0; i < myFunction.arguments.length;
        i++) {
        alert("Here's one: " +
               myFunction.arguments[i])
    }
}
```

```html
<input type="button" name="myButton" value="Press me"
    onclick="myFunction("arfie", "barfie", "snarfie")">
```

You can see from its definition that myFunction() expects just one parameter: inputValue. However, the function is being called in the onClick event handler with three values — "arfie", "barfie", and "snarfie". So, myFunction() quickly figures out how many values were actually passed by examining the value of myFunction.arguments.length. The function then proceeds to display each of the three values to the user.

Functions that Help You Create and Work with Objects

When used in conjunction with a custom-built function that defines a new type of object, the new operator enables you to create an instance (or a dozen instances) of that type of object. Here's how it works. First, take a look at a function that defines a customer object. You can see from looking at this function that every customer instance that you create will have an associated name, age, sex, and occupation:

```javascript
function customer(inputName, inputAge, inputSex, inputOccupation) {
    this.name = inputName
    this.age = inputAge
    this.sex = inputSex
    this.occupation = inputOccupation
}
```

Now that there's a "mold" available for creating customers, all you need to do is use it. You can create as many customers (okay, technically, instances of objects representing customers) as you like by using the new keyword, as you can see in the following section.

```javascript
new
```

(technically, new isn't a function at all; it's an operator. It works pretty much like a function, though, which is why it's presented in this section.)

To create two separate objects representing customers, here's what you do:
Functions that Help You Create and Work with Objects

var firstCustomer = new customer("Junior Samples", 56, "M", "car dealer")
var secondCustomer = new customer("Margaret Mannfred", 34, "F", "contractor")

this
You may have noticed the this keyword in the customer function declaration shown in the code for defining a customer object. Then again, maybe you didn’t, so here it is again:

function customer(inputName, inputAge, inputSex, inputOccupation) {
    this.name = inputName
    this.age = inputAge
    this.sex = inputSex
    this.occupation = inputOccupation
}

In the preceding example, this is shorthand for the customer() function. When the JavaScript interpreter encounters the this keyword, the interpreter already knows it's inside a function called customer(); so it automatically substitutes customer() for the this keyword. That way you don't have to keep spelling out the whole function name yourself. (It also makes the code easier to read.)

with
with is a kind of shorthand that you can use to save yourself a few keystrokes. When you want to refer to several attributes of the same object (for instance, the attributes lastModified, location, and title of the document object), instead of writing this:

document.writeln(document.lastModified)
document.writeln(document.location)
document.writeln(document.title)

you can write this:

with (document) {
    writeln(lastModified)
    writeln(location)
    writeln(title)
}

The JavaScript interpreter is satisfied with (and responds identically to) either version.

For more details about using functions in JavaScript, see JavaScript For Dummies, by yours truly, published by IDG Books Worldwide, Inc.

Methods: How an Object Behaves

This part lists every method available to you in JavaScript.

In this part...
- Getting familiar with the methods available to you in JavaScript
- Deciding how (and when) to call each method
Methods

A method is just like a function (see also Part V for more on functions). Unlike functions, however, methods access only one object's data — the object with which the method is associated. A method's name is typically a verb that describes what the method does to its associated object (for example, `blink()`, `blur()`, and `click()`). As you see in the upcoming examples, you invoke methods by specifying the fully qualified (that is, the complete) name of the object to which the method belongs.

Take the `click()` method, for example. If you have three buttons on a form — buttonOne, buttonTwo, and buttonThree, all of which support the `click()` method, how would the JavaScript interpreter translate `click()` by itself? It couldn't! You have to include the method to specify which button you want to click on, like this: `document.myForm.buttonOne.click()`.

The only exception to this format is the `window` object, which doesn't usually need to be specified explicitly when you call one of its methods.

Several of the `String` methods that you run across in this part are identical to selected HTML tags in terms of their results. For example, two ways exist to make text on a Web page appear in big font, and each way is equally effective:

- A JavaScript statement: `document.write("Here is some big text".big())`
- An HTML statement: `<BIG>Here is some big text</BIG>`

Whichever you have a choice between using an HTML tag and a JavaScript statement to perform the same task, ask yourself whether you want to perform the task based on some user input. If so, opt for the JavaScript statement; if not, the HTML tag will suffice.

abs

Use the `abs()` method of the `Math` object to return the absolute value of a number.

Syntax:

```
Math.abs(number)
```

Example:

```
var myResult = Math.abs(1)
```

acos

Use the `acos()` method of the `Math` object to return the arc cosine (in radians) of a number.

Syntax:

```
Math.acos(number)
```

Example:

```
var myResult = Math.acos(1)
```

alert

Use the `alert()` method of the `window` object to display a pop-up dialog box that contains two elements: a message that you define and an OK button.

Syntax:

```
alert("message")
```

Example:

```
alert("Your order total is "+ getOrderTotal())
```

anchor

Use the `anchor()` method of the `String` object to identify a string as an HTML anchor. After you define a string as an anchor, you can use the string as the target for a hypertext link. See also the "Link" section in Part III.

Syntax:

```
string.anchor(anchonName)
```

Example:

```
"Table Of Contents".anchor("TOC_anchor")
```

Using the `anchor()` method lets you define an anchor inside a JavaScript script; the `<A>...</A>` tag pair is the way to define an anchor in HTML. For example, you can replace the preceding example with the following HTML code:

```
<A NAME="TOC_anchor">Table of Contents</A>
```

Most programmers use HTML tags to define anchors unless they want to create a link on the fly, based on some user input. To define a link based on user input, savvy JavaScript authors use the `anchor()` method instead.
**asin**

Use the `asin()` method of the `Math` object to return the arc sine (in radians) of a number.

**Syntax:**
```
Math.asin(number)
```

**Example:**
```
Math.asin(1)
```

**atan**

Use the `atan()` method of the `Math` object to return the arc tangent (in radians) of a number.

**Syntax:**
```
Math.atan(number)
```

**Example:**
```
Math.atan(1)
```

**atan2**

Use the `atan2()` method of the `Math` object to return a numeric value for the angle (theta component) of the polar coordinate \((r, \theta)\) that corresponds to the specified Cartesian coordinate \((x, y)\). (If you understood that sentence, congratulations! You’re a rocket scientist.)

**Syntax:**
```
Math.atan2(number)
```

**Example:**
```
Math.atan2(90, 15)
```

**back**

Use the `back()` method of the `history` object to load the previous URL in the history list.

**Syntax:**
```
history.back()
```

**Example:**
```
history.back()
```

Invoking the `back()` method of the `history` object produces the same result as `history.go(-1)`. Choosing `Go<Back` from the Navigator or Internet Explorer menu also produces the same result, for that matter. If you know for sure that you only want to go back one URL, the `history.back()` method is the way to go; if you have to calculate how many URLs to go back, `history.go()` is a better choice.

**big**

Using the `big()` method of the `String` object enables you to display a string in big (size 4) font. (To specify font sizes from 1 through 7, see also “`fontsize`”, described later in this part.)

**Syntax:**
```
string.big()
```

**Example:**
```
document.write("This is gonna be big!").big()
```

Using the `big()` method of the `String` object produces the same result as surrounding text with the `<BIG>...</BIG>` HTML tag pair. Typically, your choice of which method to use depends on whether you know ahead of time what text you want to display in big font. If you know what text should appear in big font, the HTML tag works fine. On the other hand, if you need to display the text dynamically, the `string.big()` method is your best bet.

**blink**

Using the `blink()` method of the `String` object enables you to display a blinking string.

**Syntax:**
```
string.blink()
```

**Example:**
```
document.write("FREE").blink()
```

Using the `blink()` method of the `String` object produces the same result as surrounding text with the `<BLINK>...</BLINK>` HTML tag pair.

**blur**

Use the `blur()` method of the `frame`, `password`, `select`, `text`, `textarea`, or `window` objects to remove focus from those objects.
(When you click on a form element and it lets you interact with the element, that element is said to have focus; when you click somewhere else, that element is said to blur.)

At the time of this writing, Internet Explorer 3.0 doesn't support window.blur().

**Syntax:**

```javascript
frameReference.blur()
passwordName.blur()
selectName.blur()
textName.blur()
textareaName.blur()
windowReference.blur()
```

**Example:**

```javascript
parent.myFrame.blur()
document.myForm.myPassword.blur()
document.myForm.mySelectField.blur()
document.myForm.myTextField.blur()
document.myForm.myTextAreaField.blur()
mySecondWindow.blur()
```

**bold**

Using the `bold()` method of the String object enables you to display a string in bold font.

**Syntax:**

```javascript
string.bold()
```

**Example:**

```javascript
document.write("IMPORTANT".bold())
```

Using the `bold()` method produces the same result as surrounding text with the `<B>...</B>` HTML tag pair.

**ceil**

For math geniuses, use the `ceil()` (for ceiling) method of the Math object to return the smallest integer that is greater than or equal to a specified number.

**Syntax:**

```javascript
Math.ceil(number)
```

**Example:**

```javascript
document.write("The ceil of 36.25 is " + Math.ceil(36.25))
```

**charAt**

Use the `charAt()` method of the String object to return one single character of a string given a specified index.

**Syntax:**

```javascript
string.charAt(index)
```

**Example:**

```javascript
// third letter will get the value 't'
var thirdLetter = "Netscape".charAt(2)
```

**clearTimeout**

Use the `clearTimeout()` method of the frame or window objects to clear a timeout that was set by using the corresponding `setTimeOut()` method. See also "setTimeout" later in this part for specifics.

**Syntax:**

```javascript
clearTimeOut(timeoutID)
```

**Example:**

```javascript
readyYetTimer = setTimeout("alert(5 seconds have elapsed. Are you ready yet?!"), 5000)
clearTimeOut(readyYetTimer)
```

**click**

Use the `click()` method of the button, checkbox, radio, reset, and submit objects to simulate a mouse click programmatically (in other words, calling the `click()` method has the same effect as a user clicking on a button with the mouse). Clicking on a radio button selects the radio button (that is, sets the radio button's checked property to true). Clicking on a check box checks the check box (sets the check box's checked property to true).

**Syntax:**

```javascript
buttonName.click()
checkboxName.click()
radioName[index].click()
resetButtonName.click()
submitButtonName.click()
```

**Example:**

```javascript
document.myForm.myButton.click()
document.myForm.myCheckbox.click()
```
(continued)
document.myForm.myRadioGroup[0].click()  
document.myForm.myReset.click()  
document.myForm.mySubmit.click()

close (document)
Use the close() method of the document object to close an
output stream that was opened with the document.open() 
method and to force any data already sent to the document to be 
displayed. See also the “open (document)” and “open (window)”
sections later in this part.

Syntax:
document.close()

Example:
myMessageWindow=window.open('', 'messageWindow')
myMessageWindow.document.writeln('This is a message 
   window, all right!')
myMessageWindow.document.close()

close (window)
Use the close() method of the window object to close a window
that was opened with the open() method of the window object.

As you'd expect, if you leave off a specific reference to a window, 
the interpreter assumes that you want to close the current 
window — with one exception. In an event handler, you must 
specify window.close() if you want to close the current window. 
(If you leave off the specific reference to a window, the current 
document closes. Go figure.)

Syntax:
windowReference.close()

Example:
close() // closes the current window  
// closes the current window from inside an event 
handler  
window.close
// closes the window called myMessageWindow 
myMessageWindow.close()

confirm
Use the confirm() method of the window object to display a
pop-up dialog box on a Web page. The dialog box that appears
contains a message, an OK button, and a Cancel button. The
confirm() method returns true if the user clicks on the OK
button and false if the user clicks on the Cancel button.

Syntax:
confirm("message")

Example:
var submitOk = confirm("Do you really want to 
   submit the form?")

The preceding code example causes a confirmation box to pop 
up and ask the user, “Do you really want to submit the form?” If 
the user clicks on the OK button, submitOk is set to true; but if 
the user clicks on the Cancel button, submitOk is set to false.

cos
Use the cos() method of the Math object to return the cosine of a
number.

Syntax:
Math.cos(number)

Example:
Math.cos(0)

exp
Use the exp() method of the Math object to return e^{number}, where e 
is Euler's constant and number is the number argument provided 
to exp().

Syntax:
Math.exp(number)

Example:
Math.exp(1)

fixed
Using the fixed() method of the String object enables you to
display a string in fixed-pitch font. (Fixed-pitch font looks sort of
like the font from an old typewriter.)

Syntax:
string.fixed()
Example:

document.write("E = MC2".fixed())

Using the fixed() method produces the same result as surrounding text with the <TT>... </TT> HTML tag pair.

floor

Use the floor() method of the Math object to return the biggest integer that is less than or equal to a specified number.

Syntax:

Math.floor(number)

Example:

var result = Math.floor(88.78)

focus

Use the focus() method of the frame, password, select, text, textarea, or window objects to give them focus.

The focus() method is handy for helping users navigate through your form. For example, if you attempt to validate an input value and determine that the value is incorrect, you can set focus back to the input field so that the user can retype the value without first having to tab backward to it.

At the time of this writing, Internet Explorer 3.0 doesn't support window.focus().

Syntax:

frameReference.focus()
passwordName.focus()
selectName.focus()
textName.focus()
textareaName.focus()
windowReference.focus()

Example:

self.frame1.focus()
document.myForm.myPassword.focus()
document.myForm.mySelectField.focus()
document.myForm.myTextField.focus()
document.myForm.myTextareaField.focus()
mySecondWindow.focus()

fontcolor

Using the fontcolor() method of the String object enables you to display a string in any conceivable color. See also Appendix B for a complete list of predefined colors; see also "Adding Multimedia to Your Web Page" in Part IX for instructions on creating your own colors.

Syntax:

string.fontcolor(color)

Example:

// using a predefined color string
document.write("This is brown".fontcolor("brown"))

// using hexadecimal RGB triplet notation
document.write("This is fuchsia".fontcolor("F00FFF"))

Using the fontcolor() method produces the same result as surrounding text with the <FONT COLOR="brown">... </FONT> HTML tag pair. Both the fontcolor method and the <FONT>... </FONT> tag pair override the default text color defined with the TEXT attribute of the <BODY>... </BODY> tag pair.

fontsize

Using the fontsize() method of the String object enables you to display a string in one of seven font sizes (where size must be an integer between 1 and 7).

Syntax:

string.fontsize(size)

Example:

var smallSize=1
document.write("This is large".fontsize(7))
document.write("This is small".fontsize(smallSize))

test the fontsize() method produces the same result as surrounding text with the <FONT SIZE=7>... </FONT> HTML tag pair.

forward

Use the forward() method of the history object to load the next URL in the history list.
**getDate**

Syntax:

```
history.forward()
```

Example:

```
history.forward() // What could be easier?
```

Invoking the `forward()` method produces the same result as using `history.go(1)` (or choosing Go to Forward from the Navigator or Internet Explorer menu).

**getDay**

Use the `getDay()` method of a Date object to access the day of the week associated with a specified date. For example, if the Date object represents Wednesday, the 26th of June, 1996, `getDay()` returns the value 3.

Syntax:

```
dateObjectName.getDay()
```

Example:

```
var today = new Date() // gets the current date
var dayOfWeek = today.getDay()
```

**getHours**

Use the `getHours()` method of a Date object to access the hour (in 24-hour military time) associated with a specified date. For example, if the Date object represents Wednesday, the 26th of June, 1996, at 3:45 p.m., `getHours()` returns the value 15.

Syntax:

```
dateObjectName.getHours()
```

**getMinutes**

Use the `getMinutes()` method of a Date object to access the minutes associated with a specified date. For example, if the Date object represents Wednesday, the 26th of June, 1996, at 3:45 p.m., `getMinutes()` returns the value 45.

Syntax:

```
dateObjectName.getMinutes()
```

Example:

```
var today = new Date() // gets today's date
var hour = today.getHours()
```

**getMonth**

Use the `getMonth()` method of a Date object to access the month of the year associated with a specified date. For example, if the Date object represents Wednesday, the 26th of June, 1996, at 3:45 p.m., `getMonth()` returns the value 5.

For a date of June, you'd think `getMonth()` would return 6, wouldn't you? Well, unfortunately, it returns 5. That's because JavaScript counts January as month 0, February as month 1, and so on.

Syntax:

```
dateObjectName.getMonth()
```

Example:

```
var today = new Date() // gets today's date
var month = today.getMonth()
```

**getSeconds**

Use the `getSeconds()` method of a Date object to access the number of seconds in the specified date. For example, if the Date object represents Wednesday, the 26th of June, 1996, at 3:45:16 p.m., `getSeconds()` returns the value 16.

Syntax:

```
dateObjectName.getSeconds()
```
**getTime**

**Example:**
```javascript
var today = new Date() // gets the current date
var seconds = today.getSeconds()
```

**getTime**

Use the `getTime()` method of a Date object to access the number of milliseconds that have elapsed since January 1, 1970, at 00:00:00 (no, I'm not kidding!). About the only use that regular folks are going to have for this method is to assign one time to another, like this: `timeToLeave = today.getTime()` (That way you never have to actually look at and decipher the darn thing.)

Versions of Navigator earlier than 3.0 (and some early beta versions of Internet Explorer Version 3.0) tend to — well — blow up if they encounter a date earlier than January 1, 1970. So avoid giving these browsers a date like this! If you're running anything other than Navigator 3.0 and want to work with dates older than 1970, try using a more recent date (say, 1980) and then doing subtraction by using regular variables (as opposed to Date objects). **See also** the "Date" section in Part IV.

**Syntax:**
```javascript
dateObject.getName()
```

**Example:**
```javascript
var startDay = new Date() // gets the current date
var startTime = today.getTime()
```

**getTimezoneOffset**

Use the `getTimezoneOffset()` method of the Date object to get the difference in minutes between the time of the Date object and Greenwich Mean Time (GMT).

**Syntax:**
```javascript
dateObject[name].getTimezoneOffset()
```

**Example:**
```javascript
today = new Date() // gets the current date
timezoneOffset = today.getTimezoneOffset() / 60
```

**getYear**

Use the `getYear()` method of a Date object to access the year of the specified date (less 1900). For example, if the Date object represents Wednesday, the 26th of June, 1996, at 3:45:16 p.m., `getYear()` returns the value 96.

**Syntax:**
```javascript
dateObject.getName().getYear()
```

**Example:**
```javascript
var today = new Date() // gets the current date
var currentYear = today.getYear()
```

**go**

Use the `go()` method of the history object to load a URL from the history list. You can supply `go()` with one of the following two parameters:

- An integer, which the interpreter uses to count forward (positive integer) or backward (negative integer) from the current list position to find a URL to load
- A string that contains a whole URL or just part of a URL; the interpreter finds the nearest matching URL and loads it

**Syntax:**
```javascript
history.go(delta | "location")
```

**Example:**
```javascript
history.go(3) // loads the third URL forward
history.go(-2) // loads the second URL back
// loads the closest URL containing the parameter string
history.go("www.austin")
```

**indexOf**

Use the `indexOf()` method of the String object to search a specified string for the occurrence of another specified string. The `indexOf()` method accepts two parameters:

- A search value that consists of a string for which to search
- An optional index value that tells the interpreter where in the original string to begin searching for the search string (the default value is 0, which means that the search begins — where else? — at the beginning)

`indexOf()` returns the first index of the original string that matches the search string (see the following real-life example). If it can't find a match at all, it returns `-1`.

**Syntax:**
```javascript
string.indexOf(searchValue, [fromIndex])
```
Example:
```javascript
var theResult = "Can you can-can?".indexOf("can")
// theResult is set to 8
```

The string "Can you can-can?" is nothing more than an indexed list of characters. As shown in this example, the first occurrence of "can" starts at index 8 — so the `theResult` is set equal to 8.

**italics**

Using the `italics()` method of the `String` object enables you to display an italicized string.

**Syntax:**
```javascript
string.italics()
```

**Example:**
```javascript
"IMPORTANT".italics()
```

Using the `italics()` method produces the same result as surrounding text with the `<i>...</i>` HTML tag pair.

**javaEnabled()**

The `javaEnabled()` method of the `navigator` object returns `true` or `false`, depending on whether Java is enabled or disabled.

**Syntax:**
```javascript
navigator.javaEnabled()
```

**Example:**
```javascript
if (navigator.javaEnabled()) {
   alert("Java is enabled in this Navigator session.")
}
```

At the time of this writing, only Navigator 3.0 supports the `javaEnabled()` method.

**join**

The `join()` method of the `array` object joins the elements of an array into one long string. `join()` accepts an optional separator argument. If an optional separator argument is provided, it's placed between the elements inside the string. If no separator is provided, a comma is used to join the elements.

**Syntax:**
```javascript
string = arrayName.join([separator])
```

**Example:**
```javascript
var animalArray = new Array("cat", "dog", "bat", "bear")
var aString = animalArray.join(" and ")
// aString is assigned "cat and dog and bat and bear"
```

**lastIndexOf**

The `lastIndexOf()` method is practically a mirror image of the `indexOf()` method. `lastIndexOf()` starts searching from the `end` of the original string; `indexOf()` starts searching from the `beginning` of the original string.

Use the `lastIndexOf()` method of the `String` object to search a specified string backwards for the occurrence of another specified string. The `lastIndexOf()` method accepts two parameters:

- A search value that consists of a string for which to search
- An optional index value that tells the interpreter where in the original string to begin searching for the search string (the default value is length of the string minus one, which means that the interpreter begins searching at the end of the string)

The `lastIndexOf()` method returns the last index of the original string that matches the search string. If it can't find a match at all, it returns `-1`.

**Syntax:**
```javascript
string.lastIndexOf(searchValue, [fromIndex])
```

**Example:**
```javascript
var theResult = "Can you can-can?".lastIndexOf("can")
// theResult is set to 12
```

**link**

Use the `link()` method of the `String` object to create an HTML hypertext link inside a JavaScript script. Remember to use the `write()` or `writeln()` methods of the document object to display the link. (It doesn't do much good to define a link on-the-fly if no one can see it to click on it, now does it?)
Syntax:

linkText.link(hrefAttribute)

Example:

document.write("My Cool Web Page".link("http://www.fictitious.com/me/my.html"))

log

Use the log() method of the Math object to return the natural logarithm (base e) of a number.

Syntax:

Math.log(number)

Example:

Math.log(10)

max

Use the max() method of the Math object to return the greater of two numbers.

Syntax:

Math.max(number1, number2)

Example:

var taxedAmount = Math.max(amount.calcOneWay(),
                          amount.calcAnotherWay()) // pay taxes on greatest amount

min

Use the min() method of the Math object to return the lesser of two numbers.

Syntax:

Math.min(number1, number2)

Example:

var youngest = Math.min(myAge, georgesAge)

open (document)

Use the open() method of the document object to open a document for writing via the write() and writeln() methods. (Well, technically you're opening something called an output stream, but you can think of the stream as the document.) When you use the open() method, be sure to use the corresponding close() method as well. See also the "close (document)" section earlier in this part.

The open() method accepts one optional parameter, mimeType. mimeType (which refers to MIME type, or Multipurpose Internet Mail Extension) is a string that specifies the format of the data that you plan to stick into the document. Most of the time, you don't have to specify a value for this parameter; it defaults to plain old HTML, which is probably what you want. Just in case, though, the following table shows your choices:

<table>
<thead>
<tr>
<th>MIME type</th>
<th>What It Is</th>
</tr>
</thead>
<tbody>
<tr>
<td>text/html</td>
<td>HTML statements (this is the default)</td>
</tr>
<tr>
<td>text/plain</td>
<td>Plain old ASCII text with end-of-line characters</td>
</tr>
<tr>
<td>image/gif</td>
<td>An image in .GIF format</td>
</tr>
<tr>
<td>image/jpeg</td>
<td>An image in .JPG format</td>
</tr>
<tr>
<td>image/x-bitmap</td>
<td>An image in bitmap format</td>
</tr>
<tr>
<td>plugin</td>
<td>A Netscape plug-in</td>
</tr>
</tbody>
</table>

Syntax:

document.open(["mimeType"])

Example:

document.open()
document.open("image/gif")

open (window)

Use the open() method of the window object to open a new Web browser window.

Syntax:

[windowVar = ] [window].open("URL", "windowName" [, "windowFeatures"])

Example:

secondWindow = open("", "statusWindow", "scrollbars=yes, width=200, height=200")

For the last optional value shown in the syntax sample (windowFeatures), you have your choice of any or all of following features. Just make sure that you surround the entire list that you create with quotes and separate each attribute-value pair with
parse

Use the parse() method of a Date object to turn a string into a date.

Syntax:
Date.parse(dateString)

Example:
myBirthday.setTime(Date.parse("Jul 26, 1996"))

pow

Use the pow() method of the Math object to return a specified base to the specified exponent power.

Syntax:
Math.pow(base, exponent)

Example:
document.write("7 to the power of 8 is " + Math.pow(7, 8))

prompt

Use the prompt() method of the window object to display a pop-up dialog box that contains a message that you define, an input field, an OK button, and a Cancel button. The optional parameter inputDefault pre-fills the input field with the default value that you specify.

Syntax:
prompt("message", inputDefault)

Example:
prompt("What size t-shirt do you want?", "Large")

random

Use the random() method of the Math object to return a random number between 0 and 1.

Syntax:
Math.random()

Example:
var aRandomNumber = Math.random()

reload

The reload() method of the location object forces a reload of the document specified by the URL in the href property of the location object. The result of the reload() method is the same as clicking on the Reload button in Navigator (or the Refresh button in Internet Explorer).

Syntax:
windowReference.location.reload()

Example:
<input type="button" name="reloadButton" value="Reload Now" onclick="self.location.reload()" />

At the time of this writing, only Navigator 3.0 supports the reload() method.

replace

The replace() method of the location object replaces the current history entry with the specified URL.

The upshot is that after JavaScript calls the replace() method, the new URL loads, and the user can no longer navigate to the previously loaded URL by clicking on the Back button.

Syntax:
windowReference.location.replace("URL")

Example:
self.location.replace("http://home.netscape.com")

At the time of this writing, only Navigator 3.0 supports the replace() method.
reset

The reset() method of the form object simulates a user clicking on a Reset button; that is, it causes all a form's elements to be reset to their default values.

Using the reset() method can make entering multiple copies of your form easier for users. When users finish filling out your form and submit it, you can call the reset() method to reset the form so they can fill it out again from scratch.

Syntax:
```javascript
document.formName.reset()
```

Example:
```javascript
if (resetOkay) {
    document.myForm.reset()
}
```

At the time of this writing, only Navigator 3.0 supports the reset() method.

reverse

The reverse() method of the array object transposes the elements of an array; that is, the order of the elements is reversed (the first element becomes the last and the last element becomes the first).

Syntax:
```javascript
arrayName.reverse()
```

Example:
```javascript
var listOfJobs = new Array("bookie", "writer", "nurse")
listOfJobs.reverse()
// Now the array is in this order: nurse, writer, bookie
```

round

The round() method of the Math object returns the value of a specified number, rounded to the nearest integer.

Syntax:
```javascript
Math.round(number)
```

Example:
```javascript
var closeEnough = Math.round(totalPrice)
```

scroll

The scroll() method of the window object lets you scroll a window to the coordinates you specify. scroll() accepts two arguments: an x-coordinate and a y-coordinate. Both values must be integers representing the number of pixels to scroll in each direction (the x-coordinate is left-to-right, and the y-coordinate is up-and-down).

This method only executes if the page the method is on contains enough displayed data to be scrollable. Put another way, unless the page is so long that scroll bars are seen, calling scroll() won't have a visible effect.

Syntax:
```javascript
windowReference.scroll(x-coordinate, y-coordinate)
```

Example:
```javascript
if(okayToScroll) {
    self.scroll(50, 100)
}
```

At the time of this writing, only Navigator 3.0 supports the scroll() method.

select

Use the select() method of the password, text, or textarea objects to select (highlight) their respective input areas.

Syntax:
```javascript
passwordName.select()
textName.select()
textareaName.select()
```

Example:
```javascript
onBlur=document.myForm.myPassword.select() onClick="document.myForm.myTextField.select()"
if (goBack()) {
    document.myForm.myTextareaField.select()
}
```

setDate

Use the setDate() method of the Date object to set the day of the month for a specified date. The required dayValue parameter can be any integer from 1 to 31.

Syntax:
```javascript
dateObjectName.setDate(dayValue)
```
**setMinutes**

The `setMinutes()` method of the `Date` object enables you to set the minutes for a specified date. The required `minutesValue` parameter can be any integer between 0 and 60. (For any value over 60, the interpreter adds the excess onto the hours portion of the date.)

**Syntax:**
```
dateObjectName.setMinutes(minutesValue)
```

**Example:**
```
averageHeeHawEpisode.setMinutes(30)
```

---

**setMonth**

The `setMonth()` method of the `Date` object enables you to set the month for a specified date. The required `monthValue` parameter can be any integer between 0 and 11, where 0 corresponds to January and 11 corresponds to December. (For any value over 11, the interpreter adds the excess onto the years portion of the date.)

**Syntax:**
```
dateObjectName.setMonth(monthValue)
```

**Example:**
```
christmas.setMonth(11)
```

---

**setSeconds**

The `setSeconds()` method of the `Date` object enables you to set the seconds for a specified date. The required `secondsValue` parameter can be any integer between 0 and 59. (For any value over 59, the interpreter adds the excess onto the minutes portion of the date.)

**Syntax:**
```
dateObjectName.setSeconds(secondsValue)
```

**Example:**
```
pickyDate.setSeconds(12)
```

---

**setTimeout**

Use the `setTimeout()` method of the `frame` or `window` objects to evaluate an expression after a specified length of time elapses. Using the `setTimeout()` method is just like setting a timer and then doing something when the timer goes off. (An obvious application for this method is a Web game: You can start the clock ticking when a user begins the game and then pop up a message saying Game Over if the user hasn't completed the game in a certain amount of time.)

This method takes the following two parameters:

- An expression to evaluate
- A number that represents the number of milliseconds to wait before evaluating the expression
setYear

Save the return value from the setTimeout() method in a variable is a really good idea because you need to use this variable if you ever want to cancel the timer. To cancel a timer, you need to pass the variable you saved to the clearTimeout() method of the frame or window object (see the example of clearTimeout() in the following example for clarification).

**Syntax:**
```
timeoutID=setTimeout(expression, number)
```

**Example:**
```
readyYetTimer=setTimeout("alert('5 seconds have elapsed. Are you ready yet?!')", "5000")
clearTimeout(readyYetTimer)
```

**setYear**

The **setYear()** method of the Date object enables you set the year for a specified date. The required **yearValue** parameter must reflect a year between 1970 and 1999. (And because the range is so narrow, you don’t have to specify the 20th century; for example, 1985 and 85 are both perfectly acceptable entries.)

Passing a *bogus* (out-of-range) value for the **yearValue** parameter shown in the following example may not cause an immediately obvious error. Because the interpreter assumes that every date falls in the 1900s, any value not between 70 (1970) and 99 (1999) has the effect of setting the date to the last day of 1969. No fireworks, no pop-up warnings — just really goofy date calculations.

**Syntax:**
```
dateObjectName.setYear(yearValue)
```

**Example:**
```
nephewGraduation.setYear(98)
```

**sin**

Use the **sin()** method of the Math object to return the sine of a number (not the number of sins committed).

**Syntax:**
```
Math.sin(number)
```

**Example:**
```
document.write("Here is the sine of PI: " + Math.sin(Math.PI))
```

**small**

Using the **small()** method of the String object enables you display a string in small font ("small" equates to font size 2). See also the **fontsize()** method described earlier in this part.

**Syntax:**
```
string.small()
```

**Example:**
```
"This is gonna be really teensy...".small()
```

Using the **small()** method of the String object produces the same result as surrounding text with the `<SMALL>...</SMALL>` HTML tag pair.

**sort**

Use the **sort()** method of the array object to sort the elements in an array. The **sort()** method accepts an optional argument that specifies a function name to use for the sorting algorithm. If a function name isn't supplied, **sort()** converts the elements in the array to strings (if they're not already strings) and compares them in lexicographic, or alphabetic, order.

The default sorting algorithm may be sufficient for your needs — unless your array elements contain numbers. But look what happens if your **array does** contain numeric values: 60, 9, and 100 will be sorted like this — "100", "60", "9" — which is probably not what you expected. The following example contains code for a numeric sorting algorithm that, when passed to the **sort()** method, causes numbers to sort correctly.

**Syntax:**
```
arrayName.sort([compareFunction])
```

**Example:**
```
// this sort works just fine for string elements
anArray.sort()

// This function returns the lesser of two values.
// JavaScript automatically calls this function
// as many times as necessary, comparing two
// values at a time, until the entire array
// of elements is sorted.
function compareNumbers(a, b) {
    return a - b
}

anArrayOfNumbers.sort(compareNumbers)
split

Use the split() method of the String object to split a string into an array of smaller strings. split() accepts an optional separator which, if provided, is used to determine where the string is divided. If no separator is provided, the resulting array consists of one element containing the entire original string.

Syntax:
string.split([separator])

Example:
arrayOfWords = "There once was a man from Nantucket".split(" ")

Internet Explorer 3.0 does not support the split() method at the time of this writing.

sqrt

Use the sqrt() method of the Math object to return the square root of a number. If the number parameter is negative, the return value is always 0. (Some math rule or something.)

Syntax:
Math.sqrt(number)

Example:
document.write("The square root of 36 is " + Math.sqrt(36))

strike

Using the strike() method of the String object enables you to display a string with strikeovers so the text looks like it's been crossed out.

Syntax:
string.strike()

Example:
"Was $49.99".strike()

Using the strike() method produces the same result as surrounding text with the <STRIKE>...</STRIKE> HTML tag pair.

sub

Use the sub() method of the String object to display text as subscript. Using this method yields the same result as surrounding text with the HTML tag pair <SUB>...</SUB>.

Syntax:
string.sub()

Example:
document.write("H" + "2".sub() + "0 = water")

submit

Use the submit() method of the form object to submit a form (that is, to send it for processing to the server program specified in the ACTION attribute, which is defined as part of the <FORM>...</FORM> tag pair).

Syntax:
formName.submit()

Example:
document.myForm.submit()

substring

Use the substring() method of the String object to return a substring (or portion) of a specified string.

Remember that the first character of a string has index 0, not 1; so when you supply an index range of (2, 4), you're actually asking for the third and fourth characters of the string to be returned to you. Why not the third through the fifth characters? Because substring() stops one character before the second index you give it.

Syntax:
string.substring(indexA, indexB)

Example:
The following example displays the string "Woman":
document.write("Woman".substring(7, 12))

Wonder Woman
0 1 2 3 4 5 6 7 8 9 10 11
Hey, so it's not very intuitive until you see the indexed string all laid out, — but it works! If you do a great deal of text manipulation, you can get the hang of it in no time.

**sup**

Use the `sup()` method of the `String` object to display text as superscript. Using this method yields the same result as surrounding text with the HTML tag pair `<SUP>`...`</SUP>`,

**Syntax:**

`string.sup()`

**Example:**

```javascript
document.write("Scooter's Better-Than-Homemade Cinnamon Rolls" + "TM".sup())
```

**tan**

Use the `tan()` method of the `Math` object to return the tangent of a number.

**Syntax:**

`Math.tan(number)`

**Example:**

```javascript
var myTangent = Math.tan(1)
```

**toGMTString**

Use the `toGMTString()` method of the `Date` object to convert a date to a string. This method uses the Internet GMT conventions (which means that the time is returned in Greenwich Mean Time), and the format of the result that returns is slightly platform-dependent. Under Windows 95, for example, you get a string similar to the following:

```
Thu. 27 Jun 1996 16:51:24 GMT
```

**Syntax:**

`dateObjectName.toGMTstring()`

**Example:**

```javascript
var today = new Date() // gets today's date
var myDateString = today.toGMTstring()
```

**toString**

Every JavaScript object supports the `toString()` method. Use this method to represent any object as a string. If the object doesn't have a string value, `toString()` returns a string containing the type of the object. For example, `window.toString()` returns the string "[object Window]".

**toLocaleString**

Use the `toLocaleString()` method of the `Date` object to convert a date to a string that is based on local time. This method returns the correct local time, but the result is platform-dependent. On Windows 95, you get a string similar to the following:

```
06/27/96 10:51:24
```

**Syntax:**

`dateObjectName.toLocalestring()`

**Example:**

```javascript
var today = new Date() // gets today's date
var myDateString = today.toLocalestring()
```

Every time that you hear the words **platform dependent**, your skin should begin to crawl and your limbs should start to twitch uncontrollably. Platform dependence is to be avoided whenever possible because it means that you either have to redo work (yuck!) or limit your audience to people running the same platform as you (double yuck!).

Sometimes you have no way around platform dependence; but, for dates, you can try to use the `getMonth()`, `getFullYear()`, and other such methods before you resort to the `toGMTString()` and `toLocaleString()` methods.

**toLowerCase**

Use the `toLowerCase()` method of the `string` object to convert a specified string to lowercase.

**Syntax:**

`string.toLowerCase()`

**Example:**

```javascript
document.write("DON'T SHOUT, I CAN HEAR YOU JUST FINE".toLowerCase())
```
toUpperCase

Syntax:
object.toString()

Example:
document.write(document.lastModified.toString())
document.write(myFunction.toString())
document.write(myAge.toString())
document.write(Array.toString())

toUpperCase

Use the toUpperCase() method of the String object to convert
a specified string to all uppercase.

Syntax:
string.toUpperCase()

Example:
document.write("could you please speak
up?".toUpperCase())

UTC

Use the UTC() method of the Date object to return the number of
milliseconds between a specified date and January 1, 1970,
00:00:00. (UTC stands for Universal Coordinated Time, and it's
basically the same thing as GMT.) Remember that, because this is
GMT, any difference between your local time and GMT is reflected
in the return value. For example, if 6 hours are between your local
time and GMT and you put in a value of hours as 0, the return
value has hours set to 6.

Syntax:
Date.UTC(year, month, day [, hours] [, minutes] [, seconds])

Example:
GMTDate = new Date(Date.UTC(96, 7, 24, 0, 0, 0))

write

Use the write() method of the document object to write expres-
sions to the specified document.

You can use the write() method not only within the
<SCRIPT>...</SCRIPT> tags, but within event handlers,
as well. For example, this statement is perfectly legal:
onClick="anotherWindow.document.write('You clicked
it!')">

Of course, you can't write to the current document in the current
window in an event handler because the document has already
been displayed.

Syntax:
write(expression1 [, expression2] ... [, expressionN])

Example:
document.write("Here is the monthly sales total: " +
monthlyTotal)

writeln

Use the writeln() method of the document object to write any
number of expressions to the specified document. Although
writeln() appends a newline character (the HTML equivalent of
a carriage return) to the end of the displayed expressions, unless
you use the newline character within an HTML tag pair that cares
(like <PRE>...</PRE>, which preformats text), the display is
identical to that of write().

Syntax:
writeln(expression1 [, expression2] ... [, expressionN])

Example:
document.writeln("<PRE>")
document.writeln("moogoo")
document.writeln("gaipan")
document.writeln("</PRE>")
Properties (Object Data)

If you're familiar with object-oriented concepts, you know that every object, by definition, is made up of two things:

- Data (implemented as properties)
- Behavior (implemented as methods)

Part VI is devoted to methods, which represent the behaviors associated with an object; this part describes properties — the items that represent an object's data.

In this part . . .

- Accessing object properties
- Modifying object properties
- Relating object properties to their corresponding HTML tag attributes
About Properties

In object-oriented terms, properties are values that describe objects. For instance, a dog object might have the following properties: color, weight, whether or not it's pedigree, and so on.

You, the JavaScript author, set many of the properties that you see in this section when you define your objects in HTML. Unfortunately, the names of properties aren't always identical to the names of the attributes you set them to. For example, you may define \texttt{ALINK="cornflowerblue"} in your \texttt{<BODY>...</BODY>} tag, but you retrieve the value this way: \texttt{document.alinkColor}.

Three properties — name, value, and length — are presented a little differently than the other properties. Just about every object contains these three properties, and their values are very similar: \texttt{form.name} refers to the form's name, \texttt{document.name} refers to the document's name, \texttt{textarea.value} refers to a textarea object's value, and so on. So, I list these properties with no qualifiers (for instance, just plain \texttt{value} instead of \texttt{document.form.textarea.value}). Under their headings, you find a list of all the objects that support these properties and other useful stuff, like which HTML tag attribute an object property corresponds to.

\textbf{action}

Use the \texttt{action} property of the \texttt{form} object to access the value of the HTML \texttt{ACTION} attribute. This value represents the URL (usually a CGI program or a \texttt{mailto:} URL) that the form is sent to when the user clicks the Submit button. You can change the value for this property.

\texttt{document.myForm.action}

The Microsoft Internet Explorer doesn't support a value of \texttt{mailto:} for the form's \texttt{ACTION} attribute at the time of this writing.

\textbf{alinkColor}

Use the \texttt{alinkColor} property of the \texttt{document} object to access the value of the ALINK attribute. The value of ALINK represents the color that link text turns when a user \texttt{activates}, or clicks on, the link. Valid values for this property include hexadecimal RGB triplets (such as \texttt{"00000000"}) and predefined strings (such as \texttt{"red"}), as shown in Appendix B. You can change the value for this property.

\texttt{document.alinkColor}

\textbf{appCodeName}

Use the \texttt{appCodeName} property of the \texttt{navigator} object to access the code name of the version of Navigator currently in use. This property is a read-only property — you can't change it.

\texttt{navigator.appCodeName}

\textbf{applets}

The \texttt{applets} array of the \texttt{document} object contains entries for each of the applets currently loaded in a page.

\texttt{document.applets[0]}
\texttt{document.applets.length}

\textbf{appName}

Use the \texttt{appName} property of the \texttt{navigator} object to access the name of the browser currently in use. You can't change the value for this property.

\texttt{navigator.appName}

\textbf{appVersion}

Use the \texttt{appVersion} property of the \texttt{navigator} object to access the version of the browser currently in use. This property is a read-only property.

\texttt{navigator.appVersion}

\textbf{arguments}

The \texttt{arguments} array is a property of the \texttt{function} object. The elements of the \texttt{arguments} array correspond to all the arguments defined for a particular \texttt{function} object.

\texttt{myFunctionName.arguments[0]}
\texttt{myFunctionName.arguments.length}

At the time of this writing, Navigator 3.0 does not support the \texttt{function.arguments} property.
**bgColor**

Use the bgColor property of the document object to access the value of the BGCOLOR attribute defined within the `<BODY>` tag pair. The value of bgColor represents the color of the entire document background, which you can easily change.

document.bgColor

**border**

Use the border property of the image object to access or change the size of an embedded image's border (in pixels).

document.myImage.border

**checked (checkbox)**

Use the checked property of the checkbox object to see whether your user has checked a check box (clicked it on) or not (clicked it off). The value for the checked property always returns either true (for checked) or false (meaning unchecked). You can change the value of this property.

document.myForm.myCheckbox.checked

**checked (radio)**

Use the checked property of the radio object to see whether your user has set a radio button (clicked it on) or not (clicked it off). The value for the checked property always returns either true (for set) or false (not set). You can change the value of this property.

document.myForm.myRadioButton[0].checked

**cookie**

A cookie is a property of the document object. The value can be any string value (except no white space, semicolons, or commas are allowed). A JavaScript statement can change this property.

document.cookie

**defaultChecked (checkbox)**

Use the defaultChecked property of the checkbox object to see whether a check box was initially defaulted to selected. The value for the defaultChecked property always returns either true (meaning that the defaultChecked property was initially defaulted to on) or false (meaning that it wasn't). You can change the value of this property to override the default setting.

document.myForm.myCheckbox.defaultChecked

**defaultChecked (radio)**

Use the defaultChecked property of the radio object to see whether a radio button was initially defaulted to selected. The value for the defaultChecked property always returns either true (meaning that the defaultChecked property was initially defaulted to on) or false (meaning that it wasn't). You can change the value of this property to override the default setting.

document.myForm.myRadioButton[0].defaultChecked

**defaultSelected**

Use the defaultSelected property of the option object to see whether a selected option was selected by default (value is true) or not (value is false). You can change the value of this property.

document.myForm.selectMusic.options[0].defaultSelected

**defaultStatus**

Use the defaultStatus property of the window object to access the default message that appears in a window's status bar. You can change the value for this property.

window.defaultStatus

**complete**

Use the complete property of the image object to find out whether a user's Web browser has completed its attempt to load an image.

document.images[0].complete
defaultValue (password)

Use the defaultValue value of the password object to see the initially-defined default password value.

document.myForm.myPassword.defaultValue

defaultValue (text)

Use the defaultValue property of the text object to access the value of the VALUE attribute of a text object. You can change the value for this property.

document.myForm.myTextField.defaultValue

defaultValue (textarea)

Use the defaultValue property of the textarea object to access the contents of the textToDisplay textarea attribute. You can change the value for this property.

document.myForm.myTextareaField.defaultValue

description

The description property is associated with the mimeTypes object and the plugins object. When associated with the mimeTypes object, the description property represents a description of a MIME type. When associated with the plugins object, description represents a description of a plug-in.

navigator.mimeTypes[0].description
navigator.plugins[0].description

E

The E property of the Math object represents Euler's mathematical constant and the base of natural logarithms. This property is read-only.

Math.E

elements

The elements array is associated with the form object — the elements array enables you to access all the elements that have been defined for a form (button, checkbox, hidden, password, radio, reset, select, submit, text, and textarea). The array is in source code order (the first element defined is the 0th element in the array, and so forth). Elements in the array are read-only.

document.myForm.elements[0]
document.myForm.elements.length

eMBEDs

The EMBEDs array associated with the document object enables you to access all the plug-ins loaded into your Web page. Each element in the EMBEDs array corresponds to an element defined with the HTML tag <EMBED>.

document.EMBEDs[0]
document.EMBEDs.length

enabledPlugin

Use the enabledPlugin property of the mimeTypes object to access the plugins object that handles the corresponding MIME type (see navigator.mimeTypes.type).

navigator.mimeTypes[0].enabledPlugin

encoding

Use the encoding property of the form object to access the ENCTYPE form attribute. The value is one of the following two strings: application/x-www-form-urlencoded, which is the default value, or multipart/form-data. You can change the value for this property.

document.myForm.encoding

fgColor

The fgColor property of the document object describes the color of the text displayed in a document. The fgColor property corresponds to the TEXT attribute defined as part of the <BODY>...</BODY> tag pair. You can change the value for this property, but values must be either hexadecimal RGB triplets or predefined strings. See also Appendix B.

document.fgColor
filename

Use the filename property of the plugins object to access the name of the plug-in executable file on disk.

navigator.plugins[0].filename
navigator.plugins["pluginName"].filename

forms

The forms array associated with the document object enables you to access all the forms that have been defined for a document. The forms array is in source code order (the first form defined is the first element in the array ([0]), and so forth). Elements of the forms array are read-only.

document.forms[0]
document.forms.length

frames

The frames array of the window object enables you to access all the frame objects associated with a window. The frames array of the window object contains an entry for each frame object defined with the <FRAME> tag inside a <FRAMESET> tag (such frame objects are also called child frames), in the order they appear in the source code. Elements of the frames array are read-only.

frames[0]
frames.length

hash

The hash property is associated with the area, link, and location objects. In all these objects, the hash property references one little piece of the overall href property belonging to each object. The href property contains an entire URL; the hash property contains the piece of that URL that represents an anchor name. (The hash property is named hash because anchor names are always preceded by a hash symbol (#), like this: "#TOC".)

You can change the value for this property.

// syntax for link object as well as area object
document.links[index].hash
location.hash

height

The height property of the image object lets you access the value of the HEIGHT attribute of the HTML <IMG> tag. A valid value for the height property can either be an integer (measured in pixels), or a percentage (expressing a percent of total window height). The value for this property is read-only.

document.myImage.height

host

The host property is associated with the area, link, and location objects. Use the host property of each object to access the hostname:port portion of a URL. A valid value for the host property must be a string representing a hostname and a port, separated by a colon (:) in the order they appear in the source code.

A port, or more correctly, a port address, tells your browser exactly what software process on a server the browser should talk to when it loads a URL. In practice, ports are often predefined, so you may never have occasion to worry about specifying a port name.

You can change the value for this property.

// syntax for link object as well as area object

document.links[index].host
location.host

hostname

The hostname property is associated with the area, link, and location objects. The hostname property of each object refers to the host and domain name of a network host. A valid value for the hostname property is a string similar to the following:
austin.ibm.com. You can change the value for the hostname property.

// syntax for link object as well as area object

document.links[index].hostname
location.hostname

href

The href property is associated with the area, link, and location objects. The href property of each object enables you to access a string that represents the entire URL for that object.
hspace

You can change the value for this property. An example of a valid value is \texttt{http://software.ibm.com}.

// syntax for link object as well as area object
\texttt{document.links[index].href}

\texttt{location.href}

Be aware that the \texttt{href} property is comprised of several other properties (\texttt{hash}, \texttt{host}, \texttt{hostname}, \texttt{port}, and \texttt{search}). That means that if you change the values of any of these other properties, you're also indirectly changing the value of \texttt{href}.

hspace

The \texttt{hspace} property of the \texttt{image} object enables you to access the value of the HSPACE attribute of the HTML \texttt{<IMG>} tag. This value specifies a margin (measured in pixels) between the left and right edges of an image and the surrounding screen real estate. The \texttt{hspace} property only applies to images that have "left" or "right" defined for the value of the ALIGN attribute of the \texttt{<IMG>} tag.

\texttt{document.myImage.hspace}

images

The \texttt{images} array associated with the \texttt{document} object enables you to access all the images loaded into your page. Each element in the \texttt{images} array corresponds to an element defined with the HTML tag \texttt{<IMG>}.

\texttt{document.images[0]}
\texttt{document.images.length}

index

The \texttt{index} property of the \texttt{option} object enables you to access the position of an option in a \texttt{select} object. Values for the \texttt{index} property are integers, beginning with zero, and are read-only.

\texttt{document.myForm.mySelectField.options[0].index}

lastModified

Access the \texttt{lastModified} property of the \texttt{document} object to determine when a document was last modified. The value for this property is a string, and it's read-only.

\texttt{document.lastModified}

length

The \texttt{length} property is used on several different objects and arrays, and its value is always read-only.

<table>
<thead>
<tr>
<th>Object</th>
<th>Usage</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>\texttt{myArray.length}</td>
<td>Number of array elements</td>
</tr>
<tr>
<td>form</td>
<td>\texttt{document.myForm.length}</td>
<td>Number of elements for a form</td>
</tr>
<tr>
<td>frame</td>
<td>\texttt{myFrame.length}</td>
<td>Number of child frames within a frame</td>
</tr>
<tr>
<td>history</td>
<td>\texttt{history.length}</td>
<td>Number of entries in the history object</td>
</tr>
<tr>
<td>radio</td>
<td>\texttt{document.myForm.myRadio.length}</td>
<td>Number of radio buttons in a radio object</td>
</tr>
<tr>
<td>select</td>
<td>\texttt{document.myForm.mySelect.length}</td>
<td>Number of options in a select object</td>
</tr>
<tr>
<td>string</td>
<td>\texttt{document.myForm.myString.length}</td>
<td>Number of characters in a string</td>
</tr>
<tr>
<td>window</td>
<td>\texttt{length}</td>
<td>Number of frames within a window</td>
</tr>
</tbody>
</table>

Array

<table>
<thead>
<tr>
<th>Array</th>
<th>Usage</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchors</td>
<td>\texttt{document.anchors.length}</td>
<td>Number of anchors in a document</td>
</tr>
<tr>
<td>elements</td>
<td>\texttt{document.myForm.elements.length}</td>
<td>Number of elements defined for a form</td>
</tr>
<tr>
<td>embeds</td>
<td>\texttt{document.myForm.embeds.length}</td>
<td>Number of embedded plug-ins</td>
</tr>
<tr>
<td>forms</td>
<td>\texttt{document.forms.length}</td>
<td>Number of forms defined in a document</td>
</tr>
<tr>
<td>frames</td>
<td>\texttt{frames.length}</td>
<td>Number of frames in a frame/window</td>
</tr>
<tr>
<td>images</td>
<td>\texttt{document.images.length}</td>
<td>Number of images in a document</td>
</tr>
<tr>
<td>links</td>
<td>\texttt{document.links.length}</td>
<td>Number of links in a document</td>
</tr>
<tr>
<td>options</td>
<td>\texttt{document.myForm.mySelectField.options.length}</td>
<td>Number of options in a select object</td>
</tr>
</tbody>
</table>

Surprise! Even though \texttt{anchors} is an array and its elements are purportedly in source code order (that is, the 0th element corresponds to the first anchor defined in a file, the first
element to the second anchor defined, and so on), the value of `document.anchors[index]` is always null (null is geekspeak for “never existed”). What a sense of humor those JavaScript architects had, huh?

**linkColor**

Use the `linkColor` property of the `document` object to access the value of the `LINK` attribute. This value represents the color of initially displayed link text. Valid values for this property include hexadecimal RGB triplets (such as "00000000") and predefined strings (such as "red"). You can change the values for this property. See also Appendix B.

document.linkColor

**links**

The `links` array of the `document` object enables you to access all the links that have been defined for a document. The array is in source code order (the first link defined is the 0th element in the array, and so forth). Elements of the array are read-only.

document.links[0]
document.links.length

**LN2**

The `LN2` property of the `Math` object represents the natural logarithm of the number 2 (approximately 0.693). This property is read-only. (Unless you're a heavy-duty math person, my guess is you'll never feel the urge to use `LN2` in your JavaScript calculations — but please feel free!)

Math.LN2

**LN10**

The `LN10` property of the `Math` object represents the natural logarithm of 10 (approximately 2.302). This property is read-only.

Math.LN10

**location**

Use the `location` property of the `document` object to access a document's complete URL. The value for this property is read-only.

document.location

**LOG2E**

The `LOG2E` property of the `Math` object represents the base 2 logarithm (approximately 1.442). This property is read-only.

Math.LOG2E

**LOG10E**

The `LOG10E` property of the `Math` object represents the base 10 logarithm (approximately 0.434). Because the base 10 logarithm is a mathematical constant, this property is read-only.

Math.LOG10E

**lowsrc**

The `lowsrc` property of the `image` object reflects the value of the `Lowsrc` attribute defined as part of the HTML `<img>` tag. A valid value for this property is a URL for a low-resolution version of the image to load.

document.myImage.lowsrc

**method**

The `method` property of the `form` object enables you to access the value for the `METHOD` attribute defined as part of the `<FORM>...</FORM>` tag pair. A valid value for this property is a string that specifies how form field input is sent to the server when the form is submitted (either "GET" or "POST").

Take a look at Chapter 10 of *JavaScript For Dummies* (by Yours Truly) for a complete discussion of the difference between setting this property equal to "GET" and to "POST".

document.myForm.method
mimeTypes

The mimeTypes array associated with the navigator object enables you to access all the MIME types supported by the client (whether defined internally, by helper applications, or by plugins). Each element in the mimeTypes array is a mimeTypes object:

`navigator.mimeTypes[0]`

name

The name property is available on many different objects. Unless otherwise noted, the name property always represents the NAME attribute for an object.

<table>
<thead>
<tr>
<th>Object</th>
<th>Usage</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>applet</td>
<td><code>document.applets[0].name</code></td>
<td></td>
</tr>
<tr>
<td>button</td>
<td><code>document.myForm.myButton.name</code></td>
<td></td>
</tr>
<tr>
<td>checkbox</td>
<td><code>document.myForm.myCheckbox.name</code></td>
<td></td>
</tr>
<tr>
<td>fileUpload</td>
<td><code>document.myForm.myFileUpload.name</code></td>
<td></td>
</tr>
<tr>
<td>frame</td>
<td><code>myFrame.name</code></td>
<td></td>
</tr>
<tr>
<td>hidden</td>
<td><code>document.myForm.myHidden.name</code></td>
<td></td>
</tr>
<tr>
<td>image</td>
<td><code>document.myForm.myImage.name</code></td>
<td></td>
</tr>
<tr>
<td>password</td>
<td><code>document.myForm.myPassword.name</code></td>
<td></td>
</tr>
<tr>
<td>plugin</td>
<td><code>document.embeds[0].name</code></td>
<td></td>
</tr>
<tr>
<td>radio</td>
<td><code>document.myForm.myRadio[0].name</code></td>
<td>All radio buttons in a set have the same name.</td>
</tr>
<tr>
<td>reset</td>
<td><code>document.myForm.myReset.name</code></td>
<td></td>
</tr>
<tr>
<td>select</td>
<td><code>document.myForm.mySelect.name</code></td>
<td></td>
</tr>
<tr>
<td>submit</td>
<td><code>document.myForm.mySubmit.name</code></td>
<td></td>
</tr>
<tr>
<td>text</td>
<td><code>document.myForm.myText.name</code></td>
<td></td>
</tr>
<tr>
<td>textarea</td>
<td><code>document.myForm.myTextarea.name</code></td>
<td></td>
</tr>
<tr>
<td>window</td>
<td><code>window.name</code></td>
<td><code>window.name</code> is read-only; its value is that of the windowVar attribute.</td>
</tr>
<tr>
<td>options</td>
<td><code>document.myForm.mySelect.options[0].name</code></td>
<td></td>
</tr>
</tbody>
</table>

When you open a window using the open() method, you can use the opener property of the window object to access the window of the calling document:

`windowReference.opener`

At the time of this writing, Internet Explorer 3.0 doesn't support the `window.open` property.

options

The options array, associated with the select object, enables you to access all the options that have been defined for a select object. The array is in source code order (the first option defined is the 0th element in the array, and so forth). Elements of the array are read-only:

`document.myForm.selectFieldName.options[0]`

parent (frame)

The parent property of the frame object is a synonym for a frame whose frameset contains the current frame (the frame you're in when you're referencing the parent property). When the parent is a frame, the value for the parent property is the value of the NAME attribute defined as part of the `<FRAME>`...`</FRAME>` declaration. When the parent is a window, the value for the parent property is an internal reference. (That is, you can use the internal reference to refer to the properties of the window, but the internal reference is not much to look at.) The value for the parent property is read-only.

parent

parent (window)

The parent property of the window object is a synonym for a frame whose frameset contains the current window (the window you're in when you're referencing the parent property). When the
parent is a frame, the value for the parent property is the value of
the NAME attribute defined as part of the <FRAME>...</FRAME>
declaration. When the parent is a window, the value for the parent
property is an internal reference. (That is, you can use the internal
reference to refer to the properties of the window, but the internal
reference is not much to look at.) The value for the parent
property is read-only.

pathname

The pathname property is associated with the area, link, and
location objects. Use the pathname property of each object to
access the path portion of that object's URL. You can set the string
value for the pathname property.

// syntax for both link object and area object
document.links[index].pathname

document.location.pathname

PI

PI, a property of the Math object, refers to the mathematical
constant for the ratio of the circumference of a circle to its
diameter (approximately 3.1415). The value for PI is read-only.

Math.PI

plugins

The plugins array associated with the navigator object
enables you to access all the plugins (plugin objects) currently
installed on the client. Each element in the plugins array is a
plugin object.

navigator.plugins[0]

port

The port property is associated with the area, link, and
location objects. The port property of each object is a
substring, or piece, of that object's host property (which is itself a
substring of that object's href property). The port property is
the piece of the host after the colon.

// syntax is for link object as well as area object
document.links[0].port

location.port

For a given href, if no value appears for port and the value of
protocol is http://, the server assumes a default port value of
80, which is the default port for all Web-type communications.

protocol

The protocol property is associated with the area, link, and
location objects. Use the protocol property to access a
substring of the object's URL, beginning with the first character
and ending with the first colon. (A common value for protocol is
http://:) You can change the protocol property's value.

// syntax is for both link and area objects
document.links[0].protocol

location.protocol

prototype

The prototype property is available for any object that's created
with the new operator, as shown in the following table. Use the
prototype property to add your own custom properties to whole
types of objects:

// custom property called "description" is defined
// for all date objects
Date.prototype.description = null

// a new date variable is created, called "today"
today = new Date()

// the new description property is assigned a value
today.description = "Today is my birthday!"

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Array.prototype.newPropertyName = null</td>
</tr>
<tr>
<td>Date</td>
<td>Date.prototype.newPropertyName = null</td>
</tr>
<tr>
<td>Function</td>
<td>Function.prototype.newPropertyName = null</td>
</tr>
<tr>
<td>Option</td>
<td>Option.prototype.newPropertyName = null</td>
</tr>
<tr>
<td>String</td>
<td>String.prototype.newPropertyName = null</td>
</tr>
<tr>
<td>Any user-defined object</td>
<td>Animal.prototype.newPropertyName = null</td>
</tr>
</tbody>
</table>
referrer

When a user loads a linked document, the referer property of that document holds the value for the URL of the calling document. The value for referer is read-only.

document.referrer

search (link)

Used the search property of the link object to access the portion of the link URL that contains query information (that is, form input fields sent to a CGI program to be used for a document-based database search). A valid search value is a string that begins with a question mark (?) followed by any number of attribute-value pairs, each separated by an ampersand (&). You can change the value for search.

document.links[0].search

search (location)

The search property of the location object is used to access the portion of the location URL that contains query information (that is, form input fields sent to a CGI program to be used for a document-based search of a database). A valid search value is a string that begins with a question mark (?) followed by any number of attribute-value pairs, each separated by an ampersand (&). You can change the value for search.

document.location.search

Try out any of the popular Web search engines, such as Yahoo!, AltaVista, or Excite, and you can see the value of the search property being sent to the respective CGI programs — right there in either Navigator's Location: window or Internet Explorer's Address window. For instance, when I asked the AltaVista search engine to fetch all the JavaScript-related Web pages it could find, here's what appeared in my Location: window:


selected

Use the selected property of the option object to access a Boolean (true or false) value that describes whether an option in a selection has been selected. You can change the value of the selected property.

document.myForm.mySelectField.options[0].selected

selectedIndex (options)

Use the selectedIndex property of the options array to access the index of a selected option (an integer from 0 to however many options exist). If multiple options have been selected, the value of selectedIndex contains a reference only to the index of the very first option selected. The value for options.selectedIndex is the same as for select.selectedIndex, and you can change the value.

document.myForm.mySelectField.options.selectedIndex

selectedIndex (select)

Use the selectedIndex value to access the index of a select object. If multiple options have been selected, the value of selectedIndex contains a reference only to the index of the very first option selected. The value for select.selectedIndex is the same as for options.selectedIndex, and you can change the value.

document.myForm.mySelectField.selectedIndex

self (frame)

The self property of the frame object is a synonym for the current frame. (Gee, and all those philosophers spent so much time searching for the meaning of self! If only they had coded JavaScript!) The value for self is read-only.

self

self (window)

The self property of the window object is a synonym for the current window. The value for self is read-only.

self
SORT1_2

The SORT1_2 property of the Math object represents the mathematical constant for the square root of 1/2 (approximately .707). Its value can't be changed.
Math.SORT1_2

SORT2

The SORT2 property of the Math object represents the mathematical constant for the square root of two (approximately 1.414). Its value can't be changed.
Math.SORT2

csrc

The src property of the image object reflects the value of the SRC attribute defined as part of the HTML <IMG> tag. A valid value for this property is a URL for the image to be loaded.
document.myImage.src

status

The status property of the window object contains a text value that appears in the window's status bar. You can change the status property's value.
status

The status property is different from the defaultStatus property. The value of the defaultStatus property is displayed in the status bar when nothing else is going on. In contrast, the value of the status property is displayed for a specific reason — for example, in response to an onMouseOver event (see also Part VIII, “Event Handlers,” for more on events).
suffixes

The value for the suffixes property of the mimeTypes array represents a string listing, separated by commas, of all possible file suffixes (file suffixes are sometimes called file extensions) available for each corresponding MIME type.
navigator.mimeTypes[0].suffixes
target (form)

Use the target property of a form to access a string that specifies the window in which you want to receive responses from the server. The value for form.target is initially defined by the TARGET attribute as part of the <FORM>...<FORM> tag pair, but you can change it.
document.myForm.target
target (link, area)

Use the target property of a link or area object to access a string that identifies the name of the window a linked (or area mapped) document should be loaded into. The string value for link.target is initially defined by the TARGET attribute of the <A>...</A> tag pair, and the string value for area.target is initially defined by the TARGET attribute of the <AREA> tag; you can change the values for both if you wish.

// same syntax for both link and area objects
document.links[0].target
text

Use the text property of the option object to access the text that follows an <OPTION> tag that is defined as part of a select object. You can change the value for this object.
document.myForm.mySelectField.options[0].text

WARNING!

At the time of this writing, the options[0].text property is read-only in Internet Explorer 3.0.
title

The title property of the document object enables you to access the title of a document (the text that is defined between the <TITLE>...<TITLE> tag pair). The title property's value is read-only. For example:

<HEAD><TITLE>
Welcome to Bertram's Beanie Emporium
</TITLE></HEAD>
The `top` property of the `window` object refers to the topmost window that contains frames or nested framesets. Its value is read-only.

**type**

The `type` property is available on many different objects. The `type` property represents the HTML `TYPE` attribute for an object.

At the time of this writing, only Netscape Navigator Version 3.0 supports the `type` property.

<table>
<thead>
<tr>
<th>Object</th>
<th>Usage</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>button</td>
<td><code>document.myForm.myButton.type</code></td>
<td>&quot;button&quot;</td>
</tr>
<tr>
<td>checkbox</td>
<td><code>document.myForm.myCheckbox.type</code></td>
<td>&quot;checkbox&quot;</td>
</tr>
<tr>
<td>fileUpload</td>
<td><code>document.myForm.myFile.type</code></td>
<td>&quot;file&quot;</td>
</tr>
<tr>
<td>hidden</td>
<td><code>document.myForm.myHidden.type</code></td>
<td>&quot;hidden&quot;</td>
</tr>
<tr>
<td>password</td>
<td><code>document.myForm.myPassword.type</code></td>
<td>&quot;password&quot;</td>
</tr>
<tr>
<td>radio</td>
<td><code>document.myForm.myRadio[0].type</code></td>
<td>&quot;radio&quot;</td>
</tr>
<tr>
<td>reset</td>
<td><code>document.myForm.myReset.type</code></td>
<td>&quot;reset&quot;</td>
</tr>
<tr>
<td>select</td>
<td><code>document.myForm.mySelect.type</code></td>
<td>&quot;select-one&quot;, &quot;select-multiple&quot; (MULTIPLE)</td>
</tr>
<tr>
<td>submit</td>
<td><code>document.myForm.mySubmit.type</code></td>
<td>&quot;submit&quot;</td>
</tr>
<tr>
<td>text</td>
<td><code>document.myForm.myText.type</code></td>
<td>&quot;text&quot;</td>
</tr>
<tr>
<td>textarea</td>
<td><code>document.myForm.myTextarea.type</code></td>
<td>&quot;textarea&quot;</td>
</tr>
</tbody>
</table>

**type (mimeTypes)**

The `type` property of the `mimeTypes` array, instead of representing the HTML `TYPE` attribute the way it does for all of the other objects, represents one of the MIME types supported by a browser session. See also the `mimeTypes` property described earlier in this part.

```
navigator.mimeTypes[0].type
```
vlinkColor

The vlinkColor property enables you to access the value defined by the VLINK attribute declared as part of the "BODY" tag pair. The vlinkColor property defines the color of a clicked-on, or followed, link. You can change the value for the VLINK attribute to a hexadecimal RGB triplet or to a predefined color string that corresponds to such a triplet.

document.vlinkColor

vspace

The vspace property of the image object enables you to access the value of the VSPACE attribute of the HTML "IMG" tag. This value specifies a margin (measured in pixels) between the top and bottom edges of an image and the surrounding screen real estate. This property only applies to images that have "left" or "right" defined for the value of the ALIGN attribute of the "IMG" tag.

document.myImage.vspace

width

The width property of the image object lets you access the value of the WIDTH attribute of the HTML "IMG" tag. A valid value for the width property can be either an integer (measured in pixels), or a percentage (expressing a percent of total window width). The value for this property is read-only.

document.myImage.width

window

The window property is a synonym for the current window (or current frame, because frames are windows). The value for this property is read-only, and it's also the same as the self property of the window object.

Event Handlers

An event handler is a piece of code attached to some object that a Web user may interact with, like a push button, a link, a text field, and so on. When the user falls into your JavaScript-powered little Web page and does something like "clicking on a Submit button," the event handler executes some JavaScript instructions associated with that user's action. For example, when the user rolls the mouse pointer over a hypertext link, an event handler runs some code that make the URL appear in the Web browser's status bar.

This part shows you the various JavaScript event handlers that you can use in your HTML code to make your Web page more interactive and intuitive to users who visit your Web site.

In this part . . .

▷ Getting familiar with the different event handlers
▷ Understanding event handler syntax
▷ Calling event handlers
**About Events**

In JavaScript, **event** refers to some action (usually user-initiated) that affects an HTML form element. Some examples of user-initiated events include:

- Clicking (affects push buttons and radio buttons)
- Checking (affects check boxes)
- Selecting (affects text elements and list boxes)
- Changing a value (affects text and textarea elements)

You use an **event handler** to have the browser invoke a set of JavaScript statements automatically whenever the associated event occurs.

In Navigator 3.0, you can reset the function initially assigned to an event handler in an HTML statement by using a JavaScript statement. The following code snippet is an example of how you can reset the value of `onBlur` to a new function called `someOtherFunction()`:

```html
document.myForm.lastName.onBlur = "someOtherFunction"
```

Notice the lack of parentheses in the preceding assignment? (It's not `someOtherFunction()`, it's `someOtherFunction`.) When you reset the value for an event handler, you have to leave off the parentheses. Trust me, you don't want to know why.

**onAbort**

The **onAbort** event handler is associated with the **Image** object and can be used to trigger the execution of some JavaScript code each time users abort or stop loading, an image in a Web page. Users can abort an image load by clicking the Stop button on their Web browsers while the image is loading (which they typically do if the image is so large that they become impatient).

**Syntax:**

```html
onAbort="event handling text"
```

**Example:**

The **onAbort** event handler is only available on the **Image** object, as shown in the following code:

```html
<IMG NAME="thistle" SRC="images/thistle.gif"
onAbort="alert('You missed a great picture!')"
```

**onBlur**

**onBlur** is an event handler associated with the frame, select, text, textarea, and window objects. **onBlur** is invoked when an element **blurs**, or loses focus. Put another way, blurring occurs when a user clicks on one form element and then clicks on a new element, changing the focus from the first element (which is now blurred) to the new element (which is the focus of the next event, waiting to respond to user input).

**onBlur events on the select, text, and textarea objects** provide a great opportunity to do any field-level validation that you need to do. The user presumably has finished entering text or making a selection, so if you invoke a function to validate the field at this point, you can get back to the user immediately if a problem exists. *See also* "Validating User Input" in Part IX.

**Syntax:**

```html
onBlur="event handling text"
```

**Example:**

You can use the **onBlur** event handler with the frame, select, text, textarea, and window objects. The following sections contain examples of **onBlur** for each object.

**frame**

The first chunk of code below defines a frame named `frame1`, whose source is the HTML file `framcon1.html`; the second chunk of code shows how the **onBlur** event handler is implemented for `frame1`.

```html
<FRAMESET ROWS="50%,50%" COLS="40%,60%">
<FRAME SRC="framcon1.html" NAME="frame1">
<FRAME SRC="framcon2.html" NAME="frame2">
</FRAMESET>
```

(The following code, which defines the frame `frame1`, is from the file called `framcon1.html` highlighted in the preceding code sample.)

```html
<BODY BGCOLOR="lightgrey"
onBlur="document.bgColor='black'"
onFocus="document.bgColor='white'">
```

**select**

```html
<FORM>
Product: <SELECT NAME="productSelection" SIZE=1
onBlur="processSelection(this)"
```

(continued)
(continued)

```html
<OPTION VALUE="pen"> ball-point pen
<OPTION VALUE="weight"> monogrammed paperweight
</SELECT>
```

text

```
Last name: <INPUT TYPE="text" VALUE="" NAME="lastName" SIZE=25
    onBlur="verifyExistence(this.value)"
</INPUT>
```

textarea

```
<TEXTAREA NAME="commentField" ROWS=5 COLS=50
    onBlur="if (!this.value) { confirm('Are you sure you don't want to comment?') }"
</TEXTAREA>
```

window

```
<BODY BGCOLOR="lightgrey"
    onFocus="document.bgColor='white';"
    onBlur="document.bgColor='black';"
</BODY>
```

When calling a function from an event handler, passing along a

copy of the object that's involved is a great idea. You can do this

by passing the this keyword; that way, you can keep your

function code generic enough to reuse over and over again.

This code fragment, repeated from an earlier section, shows an

example of this:

```
Last name: <INPUT TYPE="text" VALUE="" NAME="lastName" SIZE=25
    onBlur="verifyExistence(this.value)"
</INPUT>
```

When a user blur's the lastName field, the JavaScript interpreter

calls the verifyExistence() function and passes it

this.value. Instead of having to know what form element value

to access in verifyExistence() (Yech, hard coding! The root

of all evil), all you have to do in verifyExistence() is work

with the generic value passed in.

**onChange**

The onChange event handler is similar to onBlur, except that

onChange is invoked only if some change has been made in the

value of a select, text, or textarea element, in addition to the

loss of focus. Depending on your application, sometimes you may

want to call the functions that do your field-level validation from

an onChange event handler instead of from onBlur. That way, a

value is verified only if it has changed, not just when a user moves

to a different area of the form. (Why go to all the trouble of

revalidating a value if you know that the value hasn't changed

since the last time that you validated it?) See also “Validating

User Input” in Part IX.

**Syntax:**

```
ochangel="event handling text"
```

**Example:**

You can use the onChange event handler with the select, text,

and textarea form elements. The following sections contain

examples of each event handler.

select

```
<FORM>
    Product: <SELECT NAME="productSelection" SIZE=1
        onChange="processSelection(this)"
        OPTION VALUE="keyring"> key ring
        <OPTION VALUE="sweatshirt"> sweatshirt
        <OPTION VALUE="horn"> monogrammed shoehorn
    </SELECT>
</FORM>
```

text

```
Last name: <INPUT TYPE="text" VALUE=""
    NAME="lastName" SIZE=25
    onBlur="verifyExistence(this.value)"
</INPUT>
```

textarea

```
<TEXTAREA NAME="commentField" ROWS=5 COLS=50
    onBlur="if (!this.value) { confirm('Are you sure you don't want to comment?') }"
</TEXTAREA>
```

**onClick**

A browser invokes the onClick event handler when the user

clicks on a clickable form element.

Navigator 3.0 enables you to stop an onClick event if your event

handler code returns false (0). For example, the following code

snippet allows users to change their minds after they click on the

link and stop the link URL from loading if they press the "Cancel"

button displayed by the confirm() method. (See also the

"confirm" section in Part VI.)

```
<A HREF="http://www.dummies.com"
    onClick="return confirm('Okay to go ahead and connect?')"
</A>
```
Internet Explorer 3.0 doesn't support this feature at the time of this writing.

**Syntax:**

```html
onClick="event handling text"
```

**Example:**

You can use the `onClick` event handler with all the following form elements: `button`, `checkbox`, `link`, `radio`, `reset`, and `submit`. The following sections contain examples of each.

**button**

```html
<input type="button" name="orderNow" value="order"
onclick="tallyOrder()"
```

**checkbox**

```html
<input type="checkbox" name="firstBook"
value="firstBook"
onclick="if (this.checked) { sendThankYouLetter() }
```

Is this your first ... For Dummies book purchase?

**link**

```html
<a href=""
onclick="this.href=pickURLBasedOnUserPreferences()"
> You'll want to see this!
```

**radio**

Which is your favorite vacation destination?

```html
<input type="radio" name="vacationSelection"
value="beach"
onclick="displayPage('beachfront property')"
The beach

<input type="radio" name="vacationSelection"
value="mountain"
onclick="displayPage('mountain cabins')"
The mountains

<input type="radio" name="vacationSelection"
value="city"
onclick="displayPage('posh hotels')"
Anywhere I can get room service
```

**reset**

```html
<input type="reset" name="reset" value="Clear form"
onclick="resetCalculatedTotals()"
```

Netscape Navigator running on a Windows platform doesn't care if you return a value of false for a `reset` `onClick` event handler; it goes ahead and resets the form no matter what happens when a user clicks the reset button — even if you think you stopped the reset from happening in the `onClick` event handling code.

**submit**

```html
<input type="submit" name="submit" value="Submit form"
onclick="checkConsistency()"
```

**onError**

You can use the `onError` event handler to trigger the execution of some JavaScript code every time an error occurs while a user is attempting to load either an image (the `Image` object) or a document (the `Window` object). A good use for the `onError` event handler is to invoke a custom function designed to examine the error, figure out what caused it, and then suggest solutions.

**Syntax:**

```html
onError="event handler text"
```

**Example:**

You can use the `onError` event handler with both the `Image` and `Window` objects. The following code shows examples of each.

**Image**

```html
<img name="noSuchAnimal" src="dummy.gif"
onerror="alert(document.myForm.noSuchAnimal.name + ' could not be loaded')"
```

**Window**

```javascript
window.onerror=myOnErrorHandler
function myOnErrorHandler(message, url, lineNumber) {
    return true
}
```

Don't want your users to see any JavaScript-generated errors for a particular document? You can head them all off at the pass before they're displayed to the user by setting the `onError` event handler equal to null, like this:

```javascript
window.onerror = null
```

**onFocus**

`onFocus` is an event handler associated with the `Frame`, `Select`, `Text`, `Textarea`, and `Window` objects. Think of this event handler as the anti-`onBlur`; instead of responding when an object loses...
focus, **onFocus** is invoked when an object gains focus. (An object gains focus when a user tabs to, or clicks on, the object.)

You might expect that selecting text in a `text` or `textarea` element would trigger the **onFocus** event handler because selecting text implies clicking on that text, which should (in turn) give focus to that element. Well, that conclusion is logical, but it's incorrect. JavaScript considers text selection within a field a special case; instead of calling the **onFocus** event handler, JavaScript invokes the **onSelect** event handler.

**See also** “**onSelect**,” later in this part.

Guess what happens if the code that’s invoked by the **onFocus** event handler contains a call to a dialog box? Well, after the element gets focus, the dialog box pops up. The user clicks OK or Cancel or whatever. Focus returns to the element. The dialog box pops up. The user clicks OK or Cancel or whatever. Focus returns to the element. The dialog box pops up. . . . Can you say *endless loop*? Please, don’t try this at home!

**Syntax:**

```plaintext```
onFocus="event handler text"
```

**Example:**

You can use the **onFocus** event handler with the `frame`, `select`, `text`, `textarea`, and `window` objects. The following sections contain examples of each.

**frame**

The first section of code below defines a frame named `fram1` whose source is the HTML file `framcon1.html`; the second section shows how the **onFocus** event handler is implemented for `fram1`.

```html```
<FRAMESET ROWS="50%,50%" COLS="40%,60%"
onLoad="display()">
    <FRAME SRC="framcon1.html" NAME="frame1">
    <FRAME SRC="framcon2.html" NAME="frame2">
</FRAMESET>
```

The following code fragment is found in the file called `framcon1.html` referenced in the preceding code.

```html```
<BODY BGCOLOR="lightgrey"
onBlur="document.bgColor='black'"
onFocus="document.bgColor='white'">
```

**onLoad**

The **onLoad** event handler is associated with both the `window` (frame) object and the `image` object. You can use the **onLoad** event handler of the `window` object in two ways:

- To trigger some JavaScript code immediately after a window loads
- To trigger some JavaScript code immediately after all the frames in a frameset load

If you define an **onLoad** event handler for both the window (in the `<BODY>... </BODY>` tag pair) and for a frameset (in the `<FRAMESET>... </FRAMESET>` tag pair), the **onLoad** event handler associated with the window executes first, and then the **onLoad** associated with the frameset executes.

**onload** gives you the opportunity to do any initialization necessary for your application — things that need to be done before the user gets a crack at the page or the image. You may want to display the current date and time, for example, or play a little welcoming tune.
**onMouseOut**

Use `onMouseOut` to recognize when a user moves the mouse pointer off a link or an area. One `onMouseOut` event is generated each time that a user moves a mouse from a link or area to someplace else on the form.

**Syntax:**

```
onMouseOut="event handler text"
```

**Example:**

`onMouseOut` is available for both the Area and link objects, as shown in the following examples.

**area**

```
<MAP NAME="thistleMap">
  <AREA NAME="topThistle" COORDS="0,0,228,318"
        HREF="javascript:displayMessage()"
        onMouseOver="self.status='When you see this message, click your left mouse button'; return true"
        onMouseOut="self.status=''; return true"
  </AREA>
</MAP>
```

**link**

```
<A HREF="http://home.netscape.com/"
   onMouseOut="status='Thanks for visiting; return true'"
   Netscape</A>
```

If you want to set the value for the status bar, as in this example, I remember to return a value of `true` in the last JavaScript statement in the handler — otherwise, the value won’t appear in the status bar.

---

**onMouseOver**

Use `onMouseOver` to recognize when a user moves the mouse pointer over a link (or over an area). One `onMouseOver` event is generated each time that a user moves a mouse to the link or area from someplace else on the form.

You can use this event handler to display a custom message in the status bar at the bottom of a document (instead of the default message) when a user drags a mouse pointer over a link or area. JavaScript automatically changes the user’s mouse pointer from an arrow into a little hand when the pointer is dragged over a link or an area.

**Syntax:**

```
onMouseOver="event handler text"
```

**Example:**

`onMouseOver` is an event handler of `area` and `link` objects. Examples of both types follow.

**area**

```
<MAP NAME="thistleMap">
  <AREA NAME="topThistle" COORDS="0,0,228,318"
        HREF="javascript:displayMessage()"
        onMouseOver="self.status='When you see this message, click your left mouse button'; return true"
        onMouseOut="self.status=''; return true"
  </AREA>
</MAP>
```

**link**

```
<A HREF="http://home.netscape.com/"
   onMouseOver="status='Visit Netscape'; return true"/>
```

Mystery Link</A>

If you want to set the value for the status bar, as in this example, remember to return a value of `true` in the last JavaScript statement in the handler. If you don’t, the interpreter ignores you and the status bar displays the default value — the URL of that link.
onReset

Use the onReset event handler of the form object to trigger some JavaScript code to execute each time that a user clicks a Reset button on a form.

Syntax:

onReset="event handler text"

Example:

The onReset event handler is only available on the form object:

```html
<FORM onReset="alert('The form values have been reset."")>
```

onSelect

Use onSelect to trigger some JavaScript code to execute each time that a user selects (highlights with the mouse) some part of the text displayed in either a text or a textarea field.

Syntax:

onSelect="event handler text"

Example:

Use the onSelect event handler with either the text or the textarea form elements. The following sections show examples of each.

**text**

Last name: \ <INPUT TYPE="text" VALUE="" NAME="lastName" SIZE=25 onSelect="showHelp()">

**textarea**

\ <TEXTAREA NAME="commentField" ROWS=5 COLS=50 onSelect="showHelp()">
\ </TEXTAREA>
\ </FORM>

In both the preceding examples, the function showHelp() is supposed to be called by the onSelect event handler when a user selects any text contained either in the lastName field or the commentField field. Unfortunately, it isn't! Netscape Navigator 3.0 (and earlier), as well as some early versions of Internet Explorer, ignores the onSelect event handler.

onSubmit

Use the onSubmit event handler of the form object to gain more control over the form-submission process. When you use a Submit object (a button that automatically submits a form when a user clicks on it), you have no way to keep the form from being submitted after the button has been clicked — even if you return false to the Submit object's onClick event handler.

A way does exist to bail out at the last minute by using the onSubmit event handler instead. All you need to do is return a value of false to onSubmit if you don't want to go through with the submit, and a value of true if you do. Often the criterion you use to make your decision is the return value of a function whose job in life is to look at the input data in its entirety to see if it's complete enough to bother submitting to the server, such as the verifyFormData() method in the example code below.

The default return value for the onSubmit event handler is true, so if you forget to return a value explicitly, the form will always be sent.

Syntax:

onSubmit="event handler text"

Example:

```javascript
function verifyFormData(incomingForm)
{
    if () // all the relevant values of incomingForm // are valid based on some criteria you // define)
    {
        return true
    }
    else {
        return false
    }
} form.onSubmit="return verifyFormData(this)"
```

onUnload

Just like its counterpart, unload, you can use the onUnload event handler associated with the window (frame) object in two ways: to trigger some JavaScript code immediately after a window has been unloaded (exited) or to trigger some JavaScript code immediately after all frames in a frameset have been unloaded. If you define an onUnload for both the window (in the <BODY>...
onUnload

</BODY> tag pair) and for a frameset (in the <FRAMESET>...<FRAMESET> tag pair), the onUnload associated with the window executes first, and then the onUnload associated with the frameset executes.

**Syntax:**

onUnload="event handler text"

**Example:**

```html
<BODY onUnload="cleanUp()">
<FRAMESET ROWS="50%,50%" COLS="40%,60%" onUnload="cleanUpFrameData()">
<FRAME SRC="frame1.html" NAME="frame1">
<FRAME SRC="frame2.html" NAME="frame2">
</FRAMESET>
```

---

**Cool Things You Can Do with JavaScript**

Everything you ever wanted to know about how to use JavaScript in your Web pages (but were afraid to ask!) is in this part. Some of the coding tips are HTML-based; others are pure JavaScript. Most of these examples show things you'll want to try out in your own Web pages; all of the examples come complete with working example code.

**In this part . . .**

- Adding multimedia to your Web page
- Integrating other components with JavaScript
- Hiding your JavaScript source code from prying eyes
- Validating user input and communicating the results to your users
Adding Multimedia to Your Web Page

You can add images, all kinds of different colors, sounds, and even animation to your Web pages.

Color

Using color is a good way to add interest and flash to your Web pages. You can specify different colors for any or all the items described in the following table. (See also Appendix B for a list of all of the predefined colors that you can use.)

<table>
<thead>
<tr>
<th>HTML &lt;BODY&gt; Attribute</th>
<th>Description</th>
<th>JavaScript Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGColor</td>
<td>Background color</td>
<td>document.bgColor</td>
</tr>
<tr>
<td>TEXT</td>
<td>Foreground color</td>
<td>document.fgColor</td>
</tr>
<tr>
<td>LINK</td>
<td>Unfollowed link color</td>
<td>document.linkColor</td>
</tr>
<tr>
<td>ALINK</td>
<td>Activated link color</td>
<td>document.alinkColor</td>
</tr>
<tr>
<td>VLINK</td>
<td>Followed link color</td>
<td>document.vlinkColor</td>
</tr>
</tbody>
</table>

Two ways exist to change the color of the elements in the preceding table: via HTML statements inside the body section of your HTML file, and via JavaScript statements. Here's an example of each approach:

```
<BODY BGColor="lime"
TEXT="darkblue"
LINK="black"
ALINK="antiquewhite"
VLINK="orange">

document.fgColor = "yellow"

But wait! There's one more way to change the color of text displayed in your Web page: the fontcolor() method of the String object. Here's how it works:

Syntax:

aString.fontcolor("someColor")

Example:

document.write("This is dark orchid.", fontcolor("darkorchid"));
```

Moving images

You can take three basic approaches to integrate animation and movies into your Web pages:

- **Create a hypertext link to an animation file:** Some popular animation file formats include .avi, .divx, .fli, .mov, .mpg, and .animated .gif. See also "Web-Surfing: Creating Hypertext Links" later in this part for the scoop on how to set up a hypertext link.

- **Add a plug-in that plays animation files:** One such plug-in is Shockwave for Director from Macromedia, which enables you to play Director movies inside your Web pages. See also "Calling Other Components from JavaScript" later in this part.

To learn more about the Shockwave for Director plug-in, visit the following URL:

http://www.macromedia.com/shockwave/

- **Use a Java applet that plays animation files:** To find out how to integrate your script with a Java applet, see also "Calling Other Components from JavaScript," later in this part.

Pictures

Inserting pictures into your Web page is brought to you courtesy of the HTML <IMG> tag. Here's an example:

```
<IMG SRC="images/mouse.gif" ALIGN="TOP"
ALT="[Rupert the Mouse]">
```

If you don't specify a path name for your image source (such as http://www.bogus.com/images/ filename.gif), your Web browser looks for the image file in the directory where your .html file is located.

Sound

Adding sounds to your Web page can be a great attention-getter, but there's a downside. Sound files (which typically have extensions .ra, .sbi, .snd, .au, or .wav) are notoriously huge, which means that users loading your sound-enabled Web page may have a long wait ahead of them. Remember, too, that not everyone on the Web is equipped with sound-playing software and speakers.

The best way to add sound to your Web page is to specify a link to a sound file. That way, users who know they aren't set up for sound (or who are in a hurry) can choose to skip your aural tidbit. See also "Web-Surfing: Creating Hypertext Links" later in this part to get all the details on creating links.
Calling Other Components from JavaScript

You can tie software components together with JavaScript to create knockout Web applications. At the time of this writing, you have three choices, depending on which Web browser you expect users to use to view your finished product: ActiveX components, Java applets, and Netscape plug-ins.

**ActiveX components**

ActiveX components are software modules that are compatible not only with Internet Explorer, but with non-Internet-related Microsoft client applications, too. The following code snippet shows how you can integrate an ActiveX component into a JavaScript script in Internet Explorer by using the `<OBJECT>` tag:

```javascript
function showProperties() {
    alert("Label properties:
    + document.myForm.spr1bl1.Angle " + " +
    + document.myForm.spr1bl1.Alignment " + " +
    + document.myForm.spr1bl1.BackStyle " + " +
    + document.myForm.spr1bl1.Caption " + " +
    + document.myForm.spr1bl1.FontName " + " +
    + document.myForm.spr1bl1.FontSize")
}
</SCRIPT>
</BODY>
</FORM NAME="myForm">

</OBJECT>
<param name="Angle" value="270">
<param name="Alignment" value="2">
<param name="BackStyle" value="0">
<param name="Caption" value="JavaScript!">
<param name="FontName" value="Times New Roman">
<param name="FontSize" value="40">
</OBJECT>

<INPUT TYPE="button" value="Look at control's properties"
NAME="showButton" onClick="showProperties()">
<br>

<INPUT TYPE="button" value="Call control's method"
NAME="callButton"
onClick="document.myForm.spr1bl1.AboutBox()">
</APPLET>
</BODY>
</HTML>

Calling Other Components from JavaScript

In the code above, the statements between the `<OBJECT>` and `</OBJECT>` tags embed an ActiveX component in a Web page. The `showButton`'s `onClick` event handler calls a function (showProperties()) that displays the embedded ActiveX component's properties to the user, while the `callButton`'s `onClick` event handler calls a method (AboutBox()) on the embedded component itself. See also the "ActiveX component" section in Part I.

At the time of this writing, only the latest version of Internet Explorer (3.01) can interact with ActiveX components, although a plug-in for Navigator is being developed that may provide ActiveX compatibility for Netscape Navigator.

Check out the following URL for a list of available ActiveX components, along with the unique programming information for each. You'll need to hook 'em up to your JavaScript scripts (like their class identifier, any parameters they require, and so forth):


For the latest details on the development of a plug-in that can make ActiveX components compatible with Navigator, visit this site:

http://www.ncompasslabs.com/products/scriptactive.htm

Java applets

You can embed Java applets into your Web pages with the help of the `<APPLET>` tag pair. See also Part I.

HTML support for Java applets is provided both by Navigator and Internet Explorer; however, JavaScript/applet interaction is currently only supported by Navigator.

Following is an example of a JavaScript statement that invokes a method called changeText() on a Java applet (check out the `onClick` event handler):

```html
<APPLET CODEBASE="http://home.netscape.com/comprod/products/navigator/version_3.0/developer"
NAME="NervousApplet" CODE="NervousText.class"
WIDTH=400 HEIGHT=50>
<PARAM NAME="text" VALUE="Enter your text here.">
</APPLET>
</BODY>
</HTML>

<INPUT TYPE="button" value="Call applet's method"
NAME="callApplet"
onClick="document.applets[0].changeText(form.InputText.value)"`
To find out the names of applet properties and methods that you can access, you can either scout out other Web pages that embed the applet, or go right to the source (the applet developer) to get official documentation.

**Netscape plug-ins**

At the time of this writing, only Navigator's implementation of JavaScript can interact with plug-ins.

Netscape plug-ins are embedded into HTML files with the `<EMBED>` tag. See also the “Plug-in” section in Part I.

Before you try this example, you need to download the Envoy document management plug-in from Tumbleweed Software and install it in Navigator’s `~/Program/plug-ins` directory on your hard drive. Here’s where you can find the Envoy plug-in:

http://www.tumbleweed.com/download.htm

The following code shows how to access an embedded Netscape plug-in’s innards via JavaScript:

```html
<HTML><HEAD><TITLE>Embedded Plug-In Example</TITLE><SCRIPT LANGUAGE="JavaScript">

function displayFlag(flagToDisplay) {
  document.envoyPlugin.setCurrentPage(flagToDisplay)
  document.envoyPlugin.executeCommand(601)
}

</SCRIPT>
</HEAD>
</BODY>
</FORM>
</BODY>
</HTML>
```

The plug-in is embedded in the page with the `<EMBED>` tag. When a user selects one of the radio buttons defined in the form (the choices are U.S., Germany, or Canada), the `displayFlag()` function is called. The values that are being passed into the `displayFlag()` function (1, 4, and 8) mean something special to the plug-in. To get hold of plug-in-specific values and methods, you can either scout out other scripts that access the plug-in, or go right to the source (the plug-in developer) to get official plug-in documentation. For instance, you wouldn’t know that the Envoy plug-in supported the methods being called from `displayFlag()` - `setCurrentPage()` and `executeCommand()` (or how to call them) unless the plug-in developer clued you in.

### Creating Your Own JavaScript Objects

As you may know, true object-oriented languages allow you to inherit from existing objects to create your own new objects (that is, base new objects on old ones so you can get away with writing less code). Because JavaScript isn’t technically object-oriented (it’s object-based), you can’t inherit from existing objects to create your own objects in JavaScript, but you can make your own objects from scratch using the new operator and a function or two. In the code example that follows, I create a virtual pet that can talk. Read on for all the exciting details.

```javascript
function talkFunction(kindOfPet) {
  if (kindOfPet == "dog") {
    document.writeln("bow-wow!")
  } else {
    if (kindOfPet == "cat") {
      document.writeln("meow-meow")
    } else {
      document.writeln("I'm speechless.")
    }
  }
}

function Pet(inputName, inputKind, inputColor) {
  this.name = inputName
  this.kind = inputKind
  this.color = inputColor
  this.speak = talkFunction(inputKind)
}
```
Given the two functions shown in the preceding example, you can create as many instances of Pet as you want:

```javascript
var myCat = new Pet("Boots", "cat", "orange striped")
var myDog = new Pet("King", "dog", "gray and brown")
var myFish = new Pet("Bubbles", "goldfish", "orange")
```

And when you want one of your virtual pets to speak, all you have to do is this:

```javascript
myCat.speak
myDog.speak
myFish.speak
```

Because myCat, myDog, and myFish contain both properties and methods (okay, one method), they're bona fide, card-carrying objects. They can even contain or be contained by other objects. For example, you may want to create an object called Person and show a relationship between your instances of Person and Pet, like this:

```javascript
function Person(inputName, inputAge, inputSex, inputPet)
    this.name = inputName
    this.age = inputAge
    this.sex = inputSex
    this.pet = inputPet

var aDog = new Pet("King", "dog", "gray and brown")
var petOwner = new Person("Bertha", "33", "Female", aDog)
alert("Here is Bertha's dog's name: " + petOwner.pet.name)
```

Take a look at the last three JavaScript statements in the code snippet above. See how an instance of Pet is being created and named aDog? Directly after that, a new Person, called petOwner, is created — and aDog (the entire object) is being passed to the Person constructor. To prove that this association worked properly, the last statement displays the value for petOwner.pet.name — which (and you can try this out for yourself) is "King."

### Displaying Scrolling Text

The following example displays scrolling text in the browser's status line, but you could display it anywhere else on the Web page just as easily.

```javascript
function displayBanner()
{
    var lengthToScroll = 85
    var rateToScroll = 85
    var displayLength = 100
    // increasing rateToScroll slows down the scroll
    lengthToScroll = (rateToScroll/message.length) + 1
    for(var i=0; i<=lengthToScroll; i++)
    {
        message += " " + message
    }
    // place it in the status bar
    status = message.substring(position, position + displayLength)
    // set new position
    if(position++ == message.length){
        position = 0
    }
    // repeat scrolling action
    id = setTimeout("displayBanner()",1000/35)
}
```

```html
<body onload="window.status=displayBanner(); return true"
    onunload="window.status=''">

As soon as the Web page is loaded, the onLoad event handler in the code above calls displayBanner(), which stuffs the message "Your message here..." in the status line. After a brief pause, displayBanner() calls itself — this time shifting the position of the displayed message slightly. For as long as the page is loaded, displayBanner() keeps calling itself every so often, shifting the message position each time so that it appears to the user as though the message is scrolling.

Use scrolling text sparingly — it can be a good way to draw attention to something on your page, but misused, it can be very annoying! Not only do many users find it distracting, but rumors are going around that over time, scrolling schemes like the preceding one can cause a user's browser to slow down or crash (depending, of course, on the version of the browser the user has installed).

### Displaying the Contents of Nested Objects

A nested object is one that is contained inside another object. For example: a document contains a form (which is why you have to refer to it as document.form), so the form is considered to be a nested object (it's nested inside a document).
Displaying the Contents of Nested Objects

You can create your own nested objects. But what if you nest objects a couple of levels deep? For debugging, you’ll probably want to be able to take a look at the contents of an entire object, including the contents of all the objects it contains (and all the objects each of them contains, and so on). The following listing contains some example code that shows you how to inspect the contents of nested objects.

```javascript
var displayString = ""
function buildDisplay(inputObject, inputObjectName){
    // These two lines set typeOfObject to the text
    // representation of inputObject
    var typeOfObject = ""
    typeOfObject += inputObject

    if (typeOfObject.substring(1,7) != "object") {
        // the object is a plain old field
        displayString += "\n" + inputObject + "\"
            + inputObjectName + "\n"
    }
    else {
        // the object contains other objects
        displayString += "Displaying object " + inputObject + "\n"

        for (var eachProperty in inputObject) {
            var nestedObjectName = ""
            nestedObjectName += inputObject[eachProperty]
            var nestedObject = inputObject[eachProperty]
            displayString += eachProperty + "=" + nestedObjectName + "\n"

            if (nestedObjectName.substring(1,7) == "object") {
                // the object contains still MORE objects
                // so call this method again (this is
                // recursion)
                buildDisplay(nestedObject, nestedObjectName)
            }
        }
    }
}
```

How does this code snippet work? Well, assume that you call `buildDisplay()` with an object called `recipeBook`, which is filled with three recipes. Each recipe, in turn, contains a name and a list of ingredients. The `buildDisplay()` function in the preceding example walks through the nested objects and displays the field values each contains, like this:

```
Displaying object recipeBook
    Displaying object recipe
    name = chocolate cake
    Displaying object listOfIngredients
    ingredientOne = flour
    ingredientTwo = sugar
    ingredientThree = eggs
    ingredientFour = cocoa
    Displaying object recipe
    name = white cake
    Displaying object listOfIngredients
    ingredientOne = flour
    ingredientTwo = sugar
    ingredientThree = eggs
    ingredientFour = vanilla
    Displaying object recipe
    name = rum cake
    Displaying object listOfIngredients
    ingredientOne = flour
    ingredientTwo = sugar
    ingredientThree = eggs
    ingredientFour = rum extract
```

Formatting Money Fields

It’s entirely possible that you may want your Web pages to display money values, and nothing looks tackier than displaying a money value as "123,44444400" (which is what JavaScript does when it’s left to its own devices). Fortunately, you have a choice — you can use the functions in the following code listing to validate and format any input field you want to constrain to money values.

```javascript
function scrubData(inputValue) {
    // This function deletes digits that aren't numbers
    // or periods. If a non-money-based digit is
    // encountered (something other than a "$", a
    // " ", or a space) an error is displayed.
    var returnValue = ""
    for (var i=0; i<inputValue.length; i++) {
        var digit = inputValue.charAt(i)
        if (parseFloat(digit) || digit == "." || digit == "0") {
            // digit is a number or period, so keep it
            returnValue += digit
        } else {
            if (digit != " " && digit != "," && digit != "$") {
                // something weird encountered
                returnValue += "ERROR"
            }
        }
    }
    return returnValue;
}
```

(continued)
else if (tempString.length == 8) {
    returnString += tempString.substring(0,3) + "" + tempString.substring(3,tempString.length-2)
}
else if (tempString.length == 9) {
    // Need to add two commas
    returnString += tempString.substring(0,1) + "," + tempString.substring(1,4) + "," + tempString.substring(4,tempString.length-2)
}
else {  // No commas necessary
    returnString = "$" + tempString.substring(0,(tempString.length-2))
}
// add in the cents
returnString += "," + tempString.substring((tempString.length - 2),tempString.length)
// Return the newly formatted string back
// to the caller
return returnString

Enter a number and click anywhere else on the page.  
You can also use the following symbols if you want: $.

<INPUT TYPE="text" NAME="desiredSalary" SIZE=15
onBlur="document.myForm.desiredSalary.value = 
asMoney(this.value)"

Here's what happens in the code above when a user types a number (say, 666666) into the desiredSalary field and then clicks elsewhere, thus blurring the field:

* asMoney() first calls scrubValue() to remove any spaces, dollar signs, commas, or periods typed in (if a user has typed in some character values, an error is displayed on-screen).
* asMoney() then adds in money punctuation by counting backwards. Two spaces back and it adds a decimal point; three spaces back and it adds a comma; finally, it tacks a dollar sign on the front.
* onBlur assigns the desiredSalary field the newly formatted value.

The result? Instead of the 666666 originally typed into the field, the value of the field post-event handler will be $666.66.
Getting Started with a Bare-Bones HTML Template

For a jump-start on your very first JavaScript-enabled Web page, try this template.

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html><head>
<title>Your Title Goes Here</title>
</head>
<body>
<script LANGUAGE="JavaScript">
/*--
// Put your JavaScript statements here
--*/
</script>
</body></html>
```

Hiding JavaScript Source Code from Users

Two different scenarios exist when hiding your JavaScript source code which might make sense. One scenario is hiding your source from users running JavaScript-enabled Web browsers. At present, it's rare that folks hide their source code from each other using this mechanism. After all, as you probably know, the legacy of the Internet is a spirit of sharing and open exchange. Still, if you want to do it, you can do so by following the example in the following section.

The second scenario, hiding your JavaScript source from users running browsers that can't interpret your scripting code anyway, is always a good idea.

From JavaScript-enabled browsers

When you create a JavaScript-enabled Web page and install it on a Web server, folks all across the world can get access to your HTML/JavaScript source. How? The same way you get access to their source code — by choosing View::Document Source (in Netscape Navigator) or View::Source (in Internet Explorer).

If you want to nip this kind of information-sharing, tree-hugging free-for-all in the bud, separate your JavaScript statements from the rest of your HTML statements and put them in a separate file with a .js filename extension. Then, in the `<SCRIPT>` tag pair, specify the name of the source file, like this:

```html
<script LANGUAGE="JavaScript" SRC="http://www.mydomain.com/myscript.js"></script>
```

Besides hiding your JavaScript source from prying eyes, bundling related JavaScript functions into one file makes it easier for you to reuse those functions in multiple Web pages. Instead of having to copy the function definitions into every HTML file in which you want to use the functions, all you have to do is reference the .js file in your Web pages, as shown above!

If you want to test this code with a file on your local machine, you can. Just set the SRC attribute equal to the name of your JavaScript file (for example, SRC="mySwellFunctions.js") and put that file in the same directory as your HTML file.

At the time of this writing, the SRC attribute is supported only by Navigator 3.0.

From non-JavaScript-enabled browsers

Users with non-JavaScript-enabled browsers who attempt to load your JavaScript-enabled Web page will be subjected to a frightfully ugly display: your JavaScript source code! (Since their browsers can't interpret JavaScript source code, the browsers assume it's text that's meant to be presented on-screen.) To keep this from happening (without affecting users running Navigator or Internet Explorer), all you have to do is add special comment characters just below the beginning `<SCRIPT>` tag and just above the ending `<SCRIPT>` tag, like so:

```html
<script LANGUAGE="JavaScript">
/*--
function someFunction() {
  ...
  */
</script>
```
Interacting with Cookies

A cookie is a chunk of browser-related (technically, client-related) information that a program on a Web server can store on the client machine. Once a cookie has been stored on a client, the next time the same server and client communicate, the server can take a look at the stored cookie and make some decisions based on it — for example, what to display to a client user and how best to display it.

The following example gives you a taste of the kind of application that cookies can help you build. It’s the Netscape PowerStart application, an application that enables users to create their own custom Web pages. PowerStart uses a JavaScript-enabled Web page, which displays a menu that offers all kinds of choices regarding Web-page color, graphics, layout, and content. Users decide which of these features they want, and when they are finished making their decisions, they click on the Build button, which saves their preferences on their machines.

After that, each time that a user selects the MyPage URL from the Netscape home page, the locally stored preferences in the file cookies.txt are extracted and used to dynamically create a custom Web page. Check out the following code listing (borrowed from the Netscape PowerStart Web page) to get a peek at how cookie-JavaScript interaction is handled:

```javascript
function SetCookie (name, thevalue) {
    // what could be simpler?
    document.cookie = name + ' = ' + thevalue + ';
    ...
    ...
}
```

Loading and Running a JavaScript Script

```javascript
function restartAll () {
    var warning = 'Sure you want to start over?
    You' +
    br() + 'will lose all of your custom settings.';

    if (confirm(warning)) {
        ...
        parent.SetCookie('pl', 'blank')
        parent.SetCookie('note', 'Reminder: Add
this page to your bookmarks list."
    }
}
```

When a user loads the PowerStart page, customizes her home page, and then clicks the Start Over button on the left-hand frame, restartAll() is called. The statements inside restartAll() call SetCookie() to “blank out” the data stored in the cookie, so the user can start again fresh.

To confirm that these things are really taking place, I encourage you to load PowerStart at URL http://personal.netscape.com /custom/modify.html and scroll to the very bottom of the left-hand frame, where you’ll see the “Start Over” button. Customize your personal Web page first, then click on Start Over and see what happens!

At the time of this writing, client-side cookie support isn’t provided in Internet Explorer 3.0.

Loading and Running a JavaScript Script

To load and run JavaScript scripts, all you have to do is load a Web page that contains embedded JavaScript statements. When you interact with the loaded page (click on a button, type in some text, that kind of thing), your browser’s built-in JavaScript interpreter recognizes the JavaScript statements and performs them for you automatically. Currently, only Netscape Navigator and Microsoft Internet Explorer support JavaScript.

There’s a slight difference between loading a JavaScript-enabled Web page that’s on your local machine and one that’s on a Web server.

<table>
<thead>
<tr>
<th>Navigator</th>
<th>Internet Explorer</th>
</tr>
</thead>
<tbody>
<tr>
<td>To load a file from your machine: File-&gt;Open File</td>
<td>File-&gt;Open</td>
</tr>
<tr>
<td>To load a file from a Web server: File-&gt;Open Location</td>
<td>File-&gt;Open</td>
</tr>
</tbody>
</table>
Looking at JavaScript Source Code

One of the best things about learning JavaScript is that when you see a really cool script, you can immediately take a behind-the-scenes look at the JavaScript and HTML code responsible. Here's how:

- If you're running Navigator and have loaded an interesting Web page, choose View→Document Source. If the page contains multiple frames, you can also click on a particular frame and select View→Frame Source to see just the code for that frame.
- If you're running Internet Explorer, choose View→Source to see a Web page's source code.

Making an Embedded Image Respond to User Events

Ever wondered how folks created those really cool clickable maps? Well, here's the secret!

```html
<MAP NAME="thistleMap">
  <AREA NAME="leftHalf" COORDS="0,0,150,300"
  HREF="javascript:displayMessage()"
  onMouseOver="self.status='Left side of the picture'; return true"
  onMouseOut="self.status=''; return true">
  <AREA NAME="rightHalf" COORDS="0,0,600,600"
  HREF="javascript:displayMessage22()"
  onMouseOver="self.status='Right side of the picture'; return true"
  onMouseOut="self.status=''; return true">
</MAP>

<IMG NAME="currentImage"
SRC="images/thistle.gif" ALIGN="MIDDLE"
ALT="[Scottish thistles]"
USEMAP="#thistleMap">
```

This code snippet produces an image that displays the text "Right side of the picture" in the status line when the user moves the mouse over the right half of the image, and displays the text "Left side of the picture" when the user (you guessed it) moves the mouse over the left half of the image.

At the time of this writing, this approach works best in Navigator 3.0. (The preceding code, interpreted by Internet Explorer 3.0, displays Shortcut to javascript:displayMessage in the status bar when a user moves a mouse over the left half of the image.)

Making Your Script Compatible with Non-JavaScript-Enabled Browsers

The quickest, lowest-common-denominator way to make your scripts presentable to non-JavaScript-enabled browsers is to follow the example in the section "Hiding JavaScript Source Code from Users" earlier in this part. The approach you'll find there ensures that browsers that can't interpret your JavaScript source code at least won't assume the source code is text and splash it all over the page for the user to see.

A more comprehensive solution is to use navigator properties to determine what brand and version of browser a user is using to load your Web page right away, before you display anything. After you know what you're dealing with, you can create two different HTML files and present the appropriate version for each kind of browser — JavaScript-friendly and non-JavaScript-friendly. More work? Absolutely. Is it worth it? It may be, if you want to dazzle the sizable portion of the global Web audience that for some reason or other hasn't yet gotten around to upgrading to a JavaScript-enabled Web browser!

```javascript
if !(navigator.appName == "Netscape" &&
navigator.appVersion.substring(0,1) == 3) {
  // The user isn't running Netscape Navigator 3.x
  // so they might not be able to see the fancy
  // stuff. If you want, you can automatically
  // load
  // a plain-vanilla version of your Web page for
  // them.
}
```

Providing Feedback to Users with Pop-Up Messages

Pop-up messages are a great way to call a user's attention to something (like, they entered the wrong value for a field). Generally, you'll want to assign one of these messages to the event handler of some input element—a button's onClick event handler, for example. Be aware, though, that pop-up messages are fairly intrusive; users have to stop everything and deal with them before they can continue with what they were doing.

JavaScript contains three different kinds of pop-up messages, which you can create using three different window methods: `alert()`, `confirm()`, and `prompt()`.
Saving JavaScript Files

When you find a really inspirational JavaScript-enabled Web page, chances are that you may want to save it so that you can study it, print it out, frame it, or maybe even (gasp!) borrow portions of it for your own Web page. You've got a choice of two formats when you save JavaScript-enabled HTML files:

- HTML source code
- WYSIWYG (what you see is what you get, or plain old text)

Saving JavaScript files as HTML source code

- If you're running Navigator 3.0, you can save HTML source by choosing File→Save As→Save as type=HTML Files.
- In Internet Explorer, it's almost the same: choose File→Save As File→Save as type=HTML (*.htm, *.html).

Whichever browser you're running, just be sure that the file name you create ends with .html or .htm.

Saving JavaScript files as text

- If you're running Navigator 3.0, you can save the text of a Web page by choosing File→Save As→Save as type=Plain Text (*.txt).
- In Internet Explorer, it's almost the same: choose File→Save As File→Save as type=Plain Text (*.txt).

Validating User Input

Choose a file name that ends with .txt, and you're home free.

Using JavaScript to Calculate Values for HTML Tags

Navigator 3.0 gives you the ability to calculate values using JavaScript statements and then assign these calculated values to HTML tag attributes. The following sample code affects the <BODY>...</BODY> tag pair, but you can assign a calculated value to any HTML attribute.

```javascript
function getRandomColor() {
    var x = Math.round(Math.random() * 255);
    return "#" + x.toString(16);
}
```

```
</SCRIPT>
</HEAD>
<BODY BGCOLOR="&{getRandomColor();}"/>
```

Each time you load a Web page that contains the preceding example code, you see a different background color. What fun!

Remember: When you assign a JavaScript statement to an HTML attribute, you need to remember to surround the JavaScript statement with an ampersand and opening curly brace (&{) in front and a closing curly brace and a semicolon (;) bringing up the rear.
Validating before the form is submitted

The onsubmit event handler of the form object is a good place to do what is called consistency editing, which is validating that the input a user types in is valid (taken as a whole). For example, a user may specify himself as “Single” in one field and enter a value for “Spouse’s name” in another field. Clearly, both values can’t be correct, because they contradict each other. It’s this type of incompatibility between values that you can check right before you pack up your form data and ship it off to a CGI program.

```javascript
function doesExist(inputValue) {
    var aCharExists = 0
    if (inputValue) {
        for (var i = 0; i < inputValue.length; i++) {
            if (inputValue.charAt(i) != " ") {
                aCharExists = 1
            }
        }
    }
    if (!aCharExists) {
        return 0
    } else {
        return 1
    }
}
```

```javascript
function editFormForConsistency() {
    if (document.myForm.single.value != "Y" &&
        doesExist (document.myForm.spouseName.value)) {
        (document.myForm.single.value == "N" &&
         !doesExist(document.myForm.spouseName.value))) {
            return false
        } else {
            return true
        }
}
```

```html
<form name="myForm" onsubmit="return editFormForConsistency()">
  Are you single? (Y or N)
  <input type="text" name="single" value="Y" size=1>
  Name of spouse: <input type="text" name="spouseName">
  <input type="submit" value="Submit!">
</form>
```

Validating one field at a time

Validating one field at a time is sometimes called field-level validation, and it’s the quickest way to give users feedback on their input. This section contains examples of the most common field-level validation criteria.

Existence:

One way to make sure that a user fills in a field is to examine the value of the field as soon as the user finishes with it to make sure that a value exists. Here’s how:

```javascript
function doesExist(inputValue, inputExpected) {
    var aCharExists = 0
    if (inputValue) {
        for (var i = 0; i < inputValue.length; i++) {
            if (inputValue.charAt(i) != " ") {
                aCharExists = 1
            }
        }
    }
    if (!aCharExists) {
        return 0
    } else {
        return 1
    }
}
```

```javascript
... Company name:
 <input type="text" name="companyName" value="" size=35
       onBlur="doesExist(this.value, 'company name')">
```

When a user clicks on the text field called companyName and then clicks somewhere else on the page, this code snippet springs into action. First, it determines whether the input value exists at all. If it does, the code loops through the input value, one character at a time, to make sure that something other than all spaces were entered. (Technically, a bunch of spaces in a row is an “existing” value to JavaScript; humans, however, generally have other ideas!) If no non-space characters were typed in, this function triggers a pop-up message to alert the user.

Numeric:

Sometimes, you may want to make sure that a user supplies a numeric value for a field (an age field, perhaps, or a price field). Depending on your needs, the JavaScript functions parseInt, parseFloat, and isNaN (all of which are discussed at length in Part V) may be just what the doctor ordered.

These functions, however, return the computer equivalent of a thumbs-up if the first digit is numeric, regardless of what the user types in after that. So, if a user types in 8abcd, parseInt,
Validating User Input

parseFloat, and isnaN would give that value their blessing! If you want to do a more thorough validation job, try the function listed in the following example.

```javascript
function isANumber(inputValue){
    if (!parseFloat(inputValue)) {
        alert("Please enter a numeric value.")
    } else {
        for (var i=0; i<inputValue.length; i++) {
            if (inputValue.charAt(i) != " ") {
                if (!parseFloat(inputValue.charAt(i)))
                    alert("Please enter a numeric value.")
                break
            }
        }
    }
}

Number of items you wish to purchase:

<input type="text" name="numberOfItems" value="" size=5
onblur="isANumber(this.value)"
 />

These JavaScript statements, repeated from the isANumber() function in the code snippet above, cause isANumber() to ignore any blank spaces that the user may accidentally type next to a number. That keeps an error from popping up if a user types in a space.

```javascript
if (inputValue.charAt(i) != " ") {
    // if we got here, the character's not a space
    if (!parseFloat(inputValue.charAt(i)))
        alert("Please enter a numeric value.")
    break
}
```
Web-Surfing: Creating Hypertext Links

To see what anchors are available on other people's Web pages, view their HTML source code. To see how to view source code, see also “Looking at JavaScript Source Code,” earlier in this part.

Creating links within a single Web page

In order to link from one part of a page to another, you need to define an anchor. An anchor is a piece of text that some other piece of text (a link) can link to. Here's an example:

```html
<H1>CENTER>How to Stay Fit in Your 100's</CENTER></H1>
<A NAME="TOC"><H2>Table of Contents</H2></A>

<A HREF="#CHAP1">Chapter 1.</A><BR>
<A HREF="#CHAP2">Chapter 2.</A><BR>
<A HREF="#CHAP3">Chapter 3.</A><BR>

<A NAME="CHAP1">Chapter 1: Aerobic Fitness
</A>

Chapter 1 text would go here, and it might be several paragraphs (even pages) long.

Chapter 2 text would go here. Pretend that this chapter is really long. What if someone read through Chapter 2 and then decided to read Chapter 1? Unless there was a link back to Chapter 1 (more likely, a link back to the Table of Contents) it would be hard for the user to scroll to the correct starting point.

Back to Table of Contents

Chapter 3 text would go here. When you're designing multiple Web pages, consider putting a button at the bottom of each page that lets the user pop back to the first page (your home page).

Back to Table of Contents

Lots of anchors are being created in the preceding code example: TOC, CHAP1, CHAP2, and CHAP3. Notice how anchors are always referenced with a hash symbol (#) in front of them? The hash symbol must always be the first character of an anchor name. See also the “Hypertext anchor” section in Part I.

Reserved Words

Most of the keywords listed in this appendix mean something special to the JavaScript interpreter. The others are reserved for future incorporation; so even though some of these words aren't used for anything right now, you still get an error if you use any of the words in this list to name variables, functions, methods, or objects.
Reserved Words

JavaScript reserved words

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

Color Values

Table B-1 contains an alphabetical listing of all the predefined colors available to you in JavaScript. As you look through the list, remember that setting a form element's color to the string "aliceblue" is equivalent to setting it equal to the RGB triplet value "#F08FF" — the choice of format is yours.

You can create your own custom colors by stringing together your very own red, green, and blue hexadecimal triplets. Here's how: In the hexadecimal scheme of things, the lowest two-digit number you can have is 00 (or 0 in decimal), the highest is FF (or 255 in decimal), and the progression for each digit looks like this:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Start with a color from the list in Table B-1 that's close to what you want — for instance, salmon. If you're looking for something with a little more red in it, increase the red portion of the value (FA) to FB. If you're looking for a shade that's just a touch less blue, reduce the blue portion of the value (72) to, say, 66. You get the idea.
### Table B-1: Predefined Color Values

<table>
<thead>
<tr>
<th>Color</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>RGB Triplet</th>
</tr>
</thead>
<tbody>
<tr>
<td>aliceblue</td>
<td>F0</td>
<td>F8</td>
<td>FF</td>
<td>FF08FF</td>
</tr>
<tr>
<td>antiquewhite</td>
<td>FA</td>
<td>EB</td>
<td>D7</td>
<td>FAE8D7</td>
</tr>
<tr>
<td>aqua</td>
<td>00</td>
<td>FF</td>
<td>FF</td>
<td>00FF00</td>
</tr>
<tr>
<td>aquamarine</td>
<td>7F</td>
<td>FF</td>
<td>D4</td>
<td>7FFFD4</td>
</tr>
<tr>
<td>azure</td>
<td>F0</td>
<td>FF</td>
<td>FF</td>
<td>F0FF00</td>
</tr>
<tr>
<td>beige</td>
<td>F5</td>
<td>F5</td>
<td>DC</td>
<td>F5F5DC</td>
</tr>
<tr>
<td>bisque</td>
<td>FF</td>
<td>E4</td>
<td>C4</td>
<td>FFE4C4</td>
</tr>
<tr>
<td>black</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>000000</td>
</tr>
<tr>
<td>blanchedalmond</td>
<td>FF</td>
<td>EB</td>
<td>CD</td>
<td>FFEBCD</td>
</tr>
<tr>
<td>blue</td>
<td>00</td>
<td>00</td>
<td>FF</td>
<td>0000FF</td>
</tr>
<tr>
<td>blueviolet</td>
<td>8A</td>
<td>2B</td>
<td>E2</td>
<td>8A2BE2</td>
</tr>
<tr>
<td>brown</td>
<td>A5</td>
<td>2A</td>
<td>2A</td>
<td>A52A2A</td>
</tr>
<tr>
<td>burlywood</td>
<td>DE</td>
<td>B8</td>
<td>87</td>
<td>DEB887</td>
</tr>
<tr>
<td>cadetblue</td>
<td>5F</td>
<td>9E</td>
<td>A0</td>
<td>5F9EA0</td>
</tr>
<tr>
<td>chartreuse</td>
<td>7F</td>
<td>FF</td>
<td>00</td>
<td>7FFFD0</td>
</tr>
<tr>
<td>chocolate</td>
<td>D2</td>
<td>69</td>
<td>1E</td>
<td>D2691E</td>
</tr>
<tr>
<td>coral</td>
<td>FF</td>
<td>7F</td>
<td>50</td>
<td>FFF7F0</td>
</tr>
<tr>
<td>cornflowerblue</td>
<td>64</td>
<td>95</td>
<td>ED</td>
<td>6495ED</td>
</tr>
<tr>
<td>cornsilk</td>
<td>FF</td>
<td>F8</td>
<td>DC</td>
<td>FFF8DC</td>
</tr>
<tr>
<td>crimson</td>
<td>DC</td>
<td>14</td>
<td>3C</td>
<td>DC143C</td>
</tr>
<tr>
<td>cyan</td>
<td>00</td>
<td>FF</td>
<td>FF</td>
<td>00FF00</td>
</tr>
<tr>
<td>darkblue</td>
<td>00</td>
<td>00</td>
<td>8B</td>
<td>00088B</td>
</tr>
<tr>
<td>darkcyan</td>
<td>00</td>
<td>8B</td>
<td>8B</td>
<td>88888B</td>
</tr>
<tr>
<td>darkgoldenrod</td>
<td>B8</td>
<td>86</td>
<td>0B</td>
<td>B8860B</td>
</tr>
<tr>
<td>darkgray</td>
<td>A9</td>
<td>A9</td>
<td>A9</td>
<td>A9A9A9</td>
</tr>
<tr>
<td>darkgreen</td>
<td>00</td>
<td>64</td>
<td>00</td>
<td>006400</td>
</tr>
<tr>
<td>darkkhaki</td>
<td>BD</td>
<td>B7</td>
<td>6B</td>
<td>BDB76B</td>
</tr>
<tr>
<td>darkmagenta</td>
<td>8B</td>
<td>00</td>
<td>8B</td>
<td>88008B</td>
</tr>
<tr>
<td>darkolivegreen</td>
<td>65</td>
<td>6B</td>
<td>2F</td>
<td>566B2F</td>
</tr>
<tr>
<td>darkorange</td>
<td>FF</td>
<td>8C</td>
<td>00</td>
<td>FF8C00</td>
</tr>
<tr>
<td>darkorchid</td>
<td>98</td>
<td>32</td>
<td>CC</td>
<td>9932CC</td>
</tr>
<tr>
<td>darkred</td>
<td>8B</td>
<td>00</td>
<td>00</td>
<td>8B0000</td>
</tr>
<tr>
<td>darksalmon</td>
<td>E9</td>
<td>96</td>
<td>7A</td>
<td>E9967A</td>
</tr>
<tr>
<td>darkseagreen</td>
<td>8F</td>
<td>BC</td>
<td>8F</td>
<td>8FB8C8</td>
</tr>
<tr>
<td>darksalmon</td>
<td>E9</td>
<td>96</td>
<td>7A</td>
<td>E9967A</td>
</tr>
<tr>
<td>darkseagreen</td>
<td>8F</td>
<td>BC</td>
<td>8F</td>
<td>8FB8C8</td>
</tr>
<tr>
<td>darkslateblue</td>
<td>4B</td>
<td>3D</td>
<td>8B</td>
<td>4B3D8B</td>
</tr>
<tr>
<td>darkslategray</td>
<td>2F</td>
<td>4F</td>
<td>4F</td>
<td>2F4F4F</td>
</tr>
<tr>
<td>darkturquoise</td>
<td>00</td>
<td>CE</td>
<td>D1</td>
<td>00CED1</td>
</tr>
<tr>
<td>darkviolet</td>
<td>94</td>
<td>00</td>
<td>D3</td>
<td>9400D3</td>
</tr>
<tr>
<td>deepink</td>
<td>FF</td>
<td>14</td>
<td>93</td>
<td>FF1493</td>
</tr>
<tr>
<td>deepskyblue</td>
<td>00</td>
<td>BF</td>
<td>FF</td>
<td>00BFF</td>
</tr>
<tr>
<td>dimgray</td>
<td>69</td>
<td>69</td>
<td>69</td>
<td>696969</td>
</tr>
<tr>
<td>dodgerblue</td>
<td>1E</td>
<td>90</td>
<td>FF</td>
<td>1E90FF</td>
</tr>
<tr>
<td>firebrick</td>
<td>B2</td>
<td>22</td>
<td>22</td>
<td>B22222</td>
</tr>
<tr>
<td>floralwhite</td>
<td>FF</td>
<td>FA</td>
<td>F0</td>
<td>FFFAF0</td>
</tr>
<tr>
<td>forestgreen</td>
<td>22</td>
<td>8B</td>
<td>22</td>
<td>228B22</td>
</tr>
<tr>
<td>fuchsia</td>
<td>FF</td>
<td>00</td>
<td>FF</td>
<td>FF00FF</td>
</tr>
<tr>
<td>gainsboro</td>
<td>DC</td>
<td>DC</td>
<td>DC</td>
<td>DCDCDC</td>
</tr>
<tr>
<td>ghostwhite</td>
<td>F8</td>
<td>F8</td>
<td>FF</td>
<td>F8F8FF</td>
</tr>
<tr>
<td>gold</td>
<td>FF</td>
<td>D7</td>
<td>00</td>
<td>FFD700</td>
</tr>
<tr>
<td>goldenrod</td>
<td>DA</td>
<td>A5</td>
<td>20</td>
<td>DAA520</td>
</tr>
<tr>
<td>gray</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>808080</td>
</tr>
<tr>
<td>green</td>
<td>00</td>
<td>00</td>
<td>80</td>
<td>008000</td>
</tr>
<tr>
<td>greenyellow</td>
<td>AD</td>
<td>FF</td>
<td>2F</td>
<td>ADDF2F</td>
</tr>
<tr>
<td>honeydew</td>
<td>F0</td>
<td>FF</td>
<td>F0</td>
<td>F0FF00</td>
</tr>
<tr>
<td>hotpink</td>
<td>FF</td>
<td>69</td>
<td>B4</td>
<td>FF69B4</td>
</tr>
<tr>
<td>indianred</td>
<td>CD</td>
<td>5C</td>
<td>5C</td>
<td>CD5C5C</td>
</tr>
<tr>
<td>indigo</td>
<td>4B</td>
<td>00</td>
<td>82</td>
<td>4B0082</td>
</tr>
<tr>
<td>ivory</td>
<td>FF</td>
<td>FF</td>
<td>F0</td>
<td>FFFFF0</td>
</tr>
<tr>
<td>khaki</td>
<td>F0</td>
<td>E6</td>
<td>8C</td>
<td>F0E68C</td>
</tr>
<tr>
<td>lavender</td>
<td>E6</td>
<td>E6</td>
<td>FA</td>
<td>E6E6FA</td>
</tr>
<tr>
<td>lavenderblush</td>
<td>FF</td>
<td>F0</td>
<td>F5</td>
<td>FFF0F5</td>
</tr>
<tr>
<td>lawngreen</td>
<td>7C</td>
<td>FC</td>
<td>00</td>
<td>7FC00</td>
</tr>
<tr>
<td>lemonchiffon</td>
<td>FF</td>
<td>FA</td>
<td>CD</td>
<td>FFFACD</td>
</tr>
<tr>
<td>lightblue</td>
<td>AD</td>
<td>D8</td>
<td>E6</td>
<td>ADD8E6</td>
</tr>
<tr>
<td>lightcoral</td>
<td>F0</td>
<td>80</td>
<td>80</td>
<td>F08080</td>
</tr>
<tr>
<td>lightcyan</td>
<td>F0</td>
<td>FF</td>
<td>FF</td>
<td>00FF00</td>
</tr>
<tr>
<td>lightgoldenrodyellow</td>
<td>FA</td>
<td>FA</td>
<td>D2</td>
<td>FAFAFD2</td>
</tr>
<tr>
<td>lightgreen</td>
<td>90</td>
<td>EE</td>
<td>90</td>
<td>90EE90</td>
</tr>
<tr>
<td>lightgrey</td>
<td>D3</td>
<td>D3</td>
<td>D3</td>
<td>D3D3D3</td>
</tr>
</tbody>
</table>

(continued)
### Color Values

<table>
<thead>
<tr>
<th>Color</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>RGB Triplet</th>
</tr>
</thead>
<tbody>
<tr>
<td>lightpink</td>
<td>FF</td>
<td>B6</td>
<td>C1</td>
<td>FFBB6C1</td>
</tr>
<tr>
<td>lightsalmon</td>
<td>FF</td>
<td>A0</td>
<td>7A</td>
<td>FFA07A</td>
</tr>
<tr>
<td>lightseagreen</td>
<td>20</td>
<td>B2</td>
<td>AA</td>
<td>20B2AA</td>
</tr>
<tr>
<td>lightskyblue</td>
<td>87</td>
<td>CE</td>
<td>FA</td>
<td>87CEFA</td>
</tr>
<tr>
<td>lightslategray</td>
<td>77</td>
<td>88</td>
<td>99</td>
<td>778899</td>
</tr>
<tr>
<td>lightsteelblue</td>
<td>B0</td>
<td>C4</td>
<td>DE</td>
<td>B0C4DE</td>
</tr>
<tr>
<td>lightyellow</td>
<td>FF</td>
<td>FF</td>
<td>E0</td>
<td>FFFFE0</td>
</tr>
<tr>
<td>lime</td>
<td>00</td>
<td>FF</td>
<td>00</td>
<td>00FF00</td>
</tr>
<tr>
<td>limegreen</td>
<td>32</td>
<td>CD</td>
<td>32</td>
<td>32CD32</td>
</tr>
<tr>
<td>linen</td>
<td>FA</td>
<td>F0</td>
<td>E6</td>
<td>FAFO6</td>
</tr>
<tr>
<td>magenta</td>
<td>FF</td>
<td>00</td>
<td>FF</td>
<td>FF00FF</td>
</tr>
<tr>
<td>maroon</td>
<td>80</td>
<td>00</td>
<td>00</td>
<td>800000</td>
</tr>
<tr>
<td>mediumaquamarine</td>
<td>66</td>
<td>CD</td>
<td>AA</td>
<td>66CDAA</td>
</tr>
<tr>
<td>mediumblue</td>
<td>00</td>
<td>00</td>
<td>CD</td>
<td>0000CD</td>
</tr>
<tr>
<td>mediumorchid</td>
<td>BA</td>
<td>55</td>
<td>D3</td>
<td>BA55D3</td>
</tr>
<tr>
<td>mediumpurple</td>
<td>93</td>
<td>70</td>
<td>DB</td>
<td>9370DB</td>
</tr>
<tr>
<td>mediumseagreen</td>
<td>3C</td>
<td>B3</td>
<td>71</td>
<td>3CB371</td>
</tr>
<tr>
<td>mediumslateblue</td>
<td>7B</td>
<td>68</td>
<td>EE</td>
<td>7B68EE</td>
</tr>
<tr>
<td>mediumspringgreen</td>
<td>00</td>
<td>FA</td>
<td>9A</td>
<td>00FA9A</td>
</tr>
<tr>
<td>mediumturquoise</td>
<td>48</td>
<td>D1</td>
<td>CC</td>
<td>48D1CC</td>
</tr>
<tr>
<td>mediumvioletred</td>
<td>C7</td>
<td>15</td>
<td>85</td>
<td>C71585</td>
</tr>
<tr>
<td>midnightblue</td>
<td>19</td>
<td>19</td>
<td>70</td>
<td>191970</td>
</tr>
<tr>
<td>mintcream</td>
<td>F5</td>
<td>FF</td>
<td>FA</td>
<td>F5FFFA</td>
</tr>
<tr>
<td>mistyrose</td>
<td>FF</td>
<td>E4</td>
<td>E1</td>
<td>FFE4E1</td>
</tr>
<tr>
<td>moccasin</td>
<td>FF</td>
<td>E4</td>
<td>B5</td>
<td>FFE4B5</td>
</tr>
<tr>
<td>navajowhite</td>
<td>FF</td>
<td>DE</td>
<td>AD</td>
<td>FFDDEAD</td>
</tr>
<tr>
<td>navy</td>
<td>00</td>
<td>00</td>
<td>80</td>
<td>000080</td>
</tr>
<tr>
<td>oldlace</td>
<td>FD</td>
<td>F5</td>
<td>E6</td>
<td>FDF5E6</td>
</tr>
<tr>
<td>olive</td>
<td>80</td>
<td>80</td>
<td>00</td>
<td>808000</td>
</tr>
<tr>
<td>olivedrab</td>
<td>6B</td>
<td>8E</td>
<td>23</td>
<td>6B8E23</td>
</tr>
<tr>
<td>orange</td>
<td>FF</td>
<td>A5</td>
<td>00</td>
<td>FFA500</td>
</tr>
<tr>
<td>orangered</td>
<td>FF</td>
<td>45</td>
<td>00</td>
<td>FF4500</td>
</tr>
<tr>
<td>orchid</td>
<td>DA</td>
<td>70</td>
<td>D6</td>
<td>DA70D6</td>
</tr>
<tr>
<td>palegoldenrod</td>
<td>EE</td>
<td>E8</td>
<td>AA</td>
<td>EE8EAA</td>
</tr>
<tr>
<td>palegreen</td>
<td>98</td>
<td>FB</td>
<td>98</td>
<td>98FB98</td>
</tr>
<tr>
<td>paleturquoise</td>
<td>AF</td>
<td>EE</td>
<td>EE</td>
<td>AFEEEE</td>
</tr>
<tr>
<td>palevioletred</td>
<td>DB</td>
<td>70</td>
<td>93</td>
<td>DB7093</td>
</tr>
<tr>
<td>papayawhip</td>
<td>FF</td>
<td>EF</td>
<td>D5</td>
<td>FFEFD5</td>
</tr>
<tr>
<td>peachpuff</td>
<td>FF</td>
<td>DA</td>
<td>B9</td>
<td>FFDAAB9</td>
</tr>
<tr>
<td>peru</td>
<td>CD</td>
<td>85</td>
<td>3F</td>
<td>CD853F</td>
</tr>
<tr>
<td>pink</td>
<td>FF</td>
<td>C0</td>
<td>CB</td>
<td>FCC0CB</td>
</tr>
<tr>
<td>plum</td>
<td>DD</td>
<td>A0</td>
<td>DD</td>
<td>DDA0DD</td>
</tr>
<tr>
<td>powderblue</td>
<td>B0</td>
<td>E0</td>
<td>E6</td>
<td>BOE0E6</td>
</tr>
<tr>
<td>purple</td>
<td>80</td>
<td>00</td>
<td>80</td>
<td>800080</td>
</tr>
<tr>
<td>red</td>
<td>FF</td>
<td>00</td>
<td>00</td>
<td>FF0000</td>
</tr>
<tr>
<td>rosybrown</td>
<td>BC</td>
<td>8F</td>
<td>8F</td>
<td>BC8F8F</td>
</tr>
<tr>
<td>royalblue</td>
<td>41</td>
<td>69</td>
<td>E1</td>
<td>4169E1</td>
</tr>
<tr>
<td>saddlebrown</td>
<td>8B</td>
<td>45</td>
<td>13</td>
<td>8B4513</td>
</tr>
<tr>
<td>salmon</td>
<td>FA</td>
<td>80</td>
<td>72</td>
<td>FA8072</td>
</tr>
<tr>
<td>sandybrown</td>
<td>F4</td>
<td>A4</td>
<td>60</td>
<td>FA4A60</td>
</tr>
<tr>
<td>seagreen</td>
<td>2E</td>
<td>8B</td>
<td>57</td>
<td>2EBB57</td>
</tr>
<tr>
<td>seashell</td>
<td>FF</td>
<td>F5</td>
<td>EE</td>
<td>FFF5EE</td>
</tr>
<tr>
<td>sienna</td>
<td>A0</td>
<td>52</td>
<td>2D</td>
<td>A0522D</td>
</tr>
<tr>
<td>silver</td>
<td>C0</td>
<td>C0</td>
<td>C0</td>
<td>C0C0C0</td>
</tr>
<tr>
<td>skyblue</td>
<td>87</td>
<td>CE</td>
<td>EB</td>
<td>87CEEB</td>
</tr>
<tr>
<td>slateblue</td>
<td>6A</td>
<td>5A</td>
<td>CD</td>
<td>6A5ACD</td>
</tr>
<tr>
<td>slategray</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>708090</td>
</tr>
<tr>
<td>snow</td>
<td>FF</td>
<td>FA</td>
<td>FA</td>
<td>FFFAFA</td>
</tr>
<tr>
<td>springgreen</td>
<td>00</td>
<td>FF</td>
<td>7F</td>
<td>00FF7F</td>
</tr>
<tr>
<td>steelblue</td>
<td>46</td>
<td>82</td>
<td>B4</td>
<td>4682B4</td>
</tr>
<tr>
<td>tan</td>
<td>D2</td>
<td>B4</td>
<td>8C</td>
<td>D2B48C</td>
</tr>
<tr>
<td>teal</td>
<td>00</td>
<td>80</td>
<td>80</td>
<td>008080</td>
</tr>
<tr>
<td>thistle</td>
<td>D8</td>
<td>BF</td>
<td>D8</td>
<td>D8BF8D</td>
</tr>
<tr>
<td>tomato</td>
<td>FF</td>
<td>63</td>
<td>47</td>
<td>FF6347</td>
</tr>
<tr>
<td>turquoise</td>
<td>40</td>
<td>E0</td>
<td>D0</td>
<td>40E0D0</td>
</tr>
<tr>
<td>violet</td>
<td>EE</td>
<td>82</td>
<td>EE</td>
<td>EE82EE</td>
</tr>
<tr>
<td>wheat</td>
<td>F5</td>
<td>DE</td>
<td>B3</td>
<td>F5DEB3</td>
</tr>
<tr>
<td>white</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FFFFFF</td>
</tr>
<tr>
<td>whitesmoke</td>
<td>F5</td>
<td>F5</td>
<td>F5</td>
<td>F5F5F5</td>
</tr>
<tr>
<td>yellow</td>
<td>FF</td>
<td>FF</td>
<td>00</td>
<td>FFFFFF</td>
</tr>
<tr>
<td>yellowgreen</td>
<td>9A</td>
<td>CD</td>
<td>32</td>
<td>9ACD32</td>
</tr>
</tbody>
</table>
**Techie Talk**

**algorithm:** An algorithm is a set of instructions designed to solve a problem. An algorithm can be expressed in English or some other human language, or translated into a computer language such as JavaScript.

**applet:** An applet is a Java program that's specifically designed to be integrated into an HTML file (that is, a Web page). On the other hand, Java programs designed to run as stand-alone applications, such as Hot Java, are called — well, they're just called Java programs.

**array:** An array is an indexed (or numbered) list of elements. You can create arrays of numbers, strings, and objects. You access an array's elements by using their indexes. For example, if you have an array called *dishes* and you want to access the first element, you type `dishes[0]`; the second element, `dishes[1]`; the third element, `dishes[2]`, and so on. The built-in arrays to which you have access in JavaScript are *anchors, elements, forms, frames, links, and options*.

**blur:** When you've clicked with your mouse pointer on a form element, that element is said to have focus. When you then click somewhere else on the Web page, that first element is said to have lost focus (*or blurred*). In JavaScript, some form elements have an *onBlur* event handler associated with them, which you can use to trigger some function when the element blurs. See *focus*.

**Boolean:** The Boolean datatype (sometimes called *logical* type) is a data type that consists of only two possible values: *true*, which is sometimes represented as 1, and *false*, which is sometimes represented as 0.

**CGI:** CGI (*Component Gateway Interface*) is a protocol that allows Web servers and Web clients to pass information back and forth to each other. CGI programs are typically written in either Perl or C languages and reside on Web servers. A client-side Web page automatically submits a form to a server-side CGI program when the user clicks on a Submit button. The CGI program is specified as part of the Web page's `<FORM>` declaration.
client: A client is a software application that makes a request, usually for data, from another software application. Although technically no law says that they have to be, client software and server software are usually located on different machines. See Web client.

content: A Web page's content is the information that the page contains, whatever that may be — an online magazine article, a product advertisement, a JavaScript example, or what have you. On the Web, authors are referred to as “content providers.”

cookie: A cookie is a piece of information about a client process (such as a Web browser session) that CGI server programs can store on the client's machine. Only the server that creates the cookie and the client that stores it have access to the information stored in the cookie. Cookies are saved in a file called cookies.txt.

daemon: A daemon is a program that runs in the background on a server machine, waiting for requests. The HyperText Transfer Protocol Daemon, or httpd, must be running on a Web server for any Web clients to be able to interact with the server.

focus: When you put your cursor on a form element and click on it, the element becomes active and lets you interact with it. That element is now said to have focus. In JavaScript, some form elements have an onFocus event handler associated with them, which you can use to trigger some function when the element receives focus. See blur.

HTML: HTML stands for HyperText Markup Language. HTML is a standard language that contains a set of conventions (called tags) that you can use to specify the appearance you want for each particular part of your Web document. HTML is supported by all Web browsers, including Navigator and Internet Explorer. All Web pages are written in HTML; some Web pages also include HTML extensions, like the <SCRIPT> tag pair that enables JavaScript script embedding.

index: An index is a number assigned to each element in an array. The first index of an array always starts with 0 and then increases by one, like so: 0, 1, 2, 3, 4, and so on.

inheritance: Inheritance is the ability of an object-oriented programming language to create an object by substantially reusing another object's characteristics — for example, the ability to create an object called Part-Time Employee from an object called Employee. JavaScript doesn't support inheritance, so it's not considered a true object-oriented language (it is considered an object-based language, though).

Internet: The Internet, or 'Net, is a worldwide computer network made up of hardware, software, and communications lines. Different protocols, like http and news, allow users to peek at content on the Internet in different ways.

interpreter: An interpreter is a piece of software that transforms human-readable source code into machine-readable language. Every JavaScript-enabled Web browser contains a JavaScript interpreter.

Java: Java is an object-oriented programming language designed by Sun Microsystems. Java is similar in some ways to C++. Java was specifically designed to be the ultimate Internet application development language, so it includes built-in features like cross-platform capability and security. You can integrate Java applets into Web pages and, with Netscape Navigator 3.0, you can integrate Java applets into JavaScript scripts.

JavaScript: JavaScript is a C-like scripting language developed by Netscape Communications. JavaScript, implemented as an extension to HTML, makes it possible for developers to create Web pages that respond to user events and perform client-side calculations. At the time of this writing, Netscape Navigator and Microsoft Internet Explorer are the only generally available browsers that support JavaScript.

LiveConnect: Implemented in Netscape Navigator Version 3.0, LiveConnect is the technology from Netscape Communications that allows JavaScript statements to interact directly with Java applets and Netscape plug-ins.

LiveWire: LiveWire (and LiveWire Pro) are server-side integrated tool suites from Netscape Communications that allow users to create, maintain, and manage Web sites. Both LiveWire and LiveWire Pro support compiled versions of JavaScript.

method: A method, sometimes called a member function, is a function that is defined specifically for a particular object and operates only on that object's data. Different objects can have methods of the same name (for example, document.close() and window.close()).

MIME type: MIME stands for Multipurpose Internet Mail Extensions. MIME types are standard types of multimedia files that can be passed around the Web. Some examples are audio/x-wav, image/jpeg, and text/html.

object: In object-oriented programming languages (or object-based languages like JavaScript), an object is a complex data type that represents a real-world person, place, thing, or idea. Objects can be built-in, like the button, check box, document, radio, and window objects, or they can be custom-defined by a programmer/JavaScript author.

polymorphism: Polymorphism, along with inheritance, is one of the defining characteristics of object-oriented languages. Polymorphism is the ability to call a method with the same name on multiple objects (for example, document.close() and window.close(), in which the close() method can be called on both the document and the window objects). JavaScript supports polymorphism.

property: A property is a descriptive characteristic of an object. For example, a Cat object might logically contain name, size, color, age, and hasShots properties, with corresponding property values of "Inky", "small", "black", "3 months", "yes". All object properties are accessible via JavaScript statements.

public: The public keyword is used in some object-oriented languages to indicate that a particular property or method defined in one class can be called from another class. Public properties and methods of Java applets can be invoked directly from JavaScript statements with Netscape Navigator 3.0.

script: A script is a simple, interpreted computer program that falls about midway on the continuum between command line directives and full-blown programming languages. JavaScript is an example of a scripting language.

server: A server is a software application that answers requests (usually for data) from other software applications, called clients. Clients and servers are typically (but not always) on different machines. See client and Web server.

source file: A human-readable computer language file that must be either compiled or interpreted in order to run on a computer. JavaScript scripts are embedded into HTML source files and are then interpreted by the HTML interpreter that's integrated into every JavaScript-enabled Web browser. See interpreter.

state: The content of an object at any point in time is referred to as the object's state.

string: A string is a collection of characters treated as an entity, usually surrounded by either single or double quotes. The String object is built-in to JavaScript and contains such methods as bold() and blink().

substring: A substring is some portion of a string.

syntax: Syntax refers to the rules of punctuation and word order of a language. JavaScript, because it's considered a scripting language (a scripting language, has its own syntax that must be followed precisely in order to create working JavaScript scripts).

tag: In HTML, a tag is a keyword like <TITLE>, <TTITLE>, <BODY>, and (</BODY>) that tells the HTML interpreter how to interpret and handle a piece of text. HTML is sometimes referred to as a tag language.
Web server: A Web server is a computer, typically running UNIX. A Web server has something called httpd (HyperText Transfer Protocol Daemon) installed and running on the server. Web servers store Web pages and CGI programs, which are accessed and used by Web clients.

Web site: A Web site is a collection of linked Web pages, produced by the same individual or company.

wizard: A wizard is a software utility that greatly speeds the completion of a task. A wizard questions a user and then automatically does something based on the user's answers. An example of a wizard is the Netscape Powerstart Setup page, which lets you choose content and layout for your own customized home page and then constructs the page for you.

WYSIWYG: What You See Is What You Get (pronounced "wizzie-wig"). WYSIWYG HTML editors, such as Microsoft Internet Assistant, are Web page editors that let you drag and drop text to create HTML files, instead of having to type in the HTML tags yourself.

Index

A
<A> tag, 19-21
abs() method, 92
accessing variables, 53
acos() method, 93
ACTION attribute of <FORM> tag, 34
action property, 126
ActiveX component, 15-16, 166-167
addition (+),
JavaScript mathematical operator, 49
JavaScript string operator, 52
addition (\=\=), JavaScript assignment operator, 47
Adobe Acrobat plug-in, 33
alert() method, 93, 181-182
algorithm, 197
aliases, associated with objects, 39
ALIGN attribute, value for, 30
ALINK (activated link) color, 164
alinkColor property, 126
alphabetical order, 117
ampersand (&), in search values, 142
anchor() method, 93
anchors
defining, 93
defining with HTML, 19-20
described, 56
in hypertext links, 187, 188
JavaScript syntax, 38
length property, 135-136
names for, 132
and (\&,\&), JavaScript logical operator, 49, 50
animated multicolored bubbles, 30-31
animation files, 165
appCodeName property, 127
append (+\=), JavaScript string operator, 52
<APPLET> tag, 29-31
applets
described, 57, 197
embedding in Web pages, 29-31
JavaScript syntax, 38
name property, 138
names of, 168
playing animation, 165
in Web pages, 167-168
applet property, 127
appName property, 127
appVersion property, 127
arc cosine
See acos() method
arc sine
See asin() method
arc tangent
See atan() method
area objects
described, 57-58
properties, 132-134, 140-141, 145
-IMG- tag and, 51
JavaScript syntax, 38
event handlers, 158-159
arguments property, 127
Array utility objects
described, 74-75, 197
JavaScript syntax, 39
join() method, 106-107
length property, 135-136
name property, 138
reverse() method, 112
sort() method, 117
asin() method, 94
assignment operators
precedence order, 50
types of, 47-48
atan() method, 94
atan2() method, 94
B
back() method, 94-95
background color, in Web pages, 164
background page color, 11
backslash\"double quote ("), escaping the quote action, 41
Bare-Bones HTML Template, 10, 18, 176
bgColor property, 128
big() method, 95
blink() method, 95
blur() method, 95-96
blurring, 151, 197
body section, defining with HTML, 11
-BODY- tag, 11-12, 183
bold() method, 96
bookmarking Web pages, 34
Boolean (true or false) data type, syntax, 37
Boolean values, 48, 198
border property, 128
brackets (\[\]), 6
break keyword, 44
browsers, 177, 181
applets property, 127
cookie property, 129
described, 59–60
embeds Array utility objects, 131
gIColor property, 131
forms array, 132
images array, 134
lastModified property, 134
linkColor property, 136
links array, 136
location property, 137
referrer property, 142
title property, 145
double quote ("), surrounding string values, 40
double slash (/)
HTML comments, 18
JavaScript comments, 42
type property, 146
type property, 147

E
E property, 130
elements array, 130–131
<EMBED> tag, 32–33, 168, 169
embeds array, 131
enabledPlugin property, 131
encoding property, 131
ENCTYPE form attribute, 131
endless loop, 156
Envoy document management plug-in,
Tumbleweed Software, 168
equal to (=)
JavaScript comparison operator, 48
JavaScript string operator, 52
equality operators, precedence order, 50
escape() function, 84
escaping quotes, 41
Euler's constant, 99
eval() function, 84–85
event handlers, 149–150
See also names of specific event
event handlers; objects
exceptions to rules, continue keyword, 44
existence, field-level validation, 185
exp() method, 99

F
fgColor property, 131
field-level validation, 151–152, 185, 186, 187
file extensions, 144
filename property, 132
fileUpload objects
defining for interactive forms, 23
described, 60–61
JavaScript syntax, 38
name property, 138

G
Build button, Netscape PowerStart, 178
buildDisplay() function, 172
built-in functions, 84–87
buttons
click() method, 97
described, 58–59
JavaScript syntax, 38
name property, 138
onClick event handler, 154
syntax for interactive forms, 22
type property, 146
value property, 147

C
calculating values for HTML tags, 183
calling functions, 82
capitalization, 41–42
Cartesian coordinate
See atan2() method
cell() method, 96
CGI (Common Gateway Interface)
protocol
described, 198
query and, 142
specifying with ACTION attribute of
<FORM> tag, 34
validating user input, 184
changeText() method, 167
charAt() method, 97
checkboxes
checked property, 128
click() method, 97
defaultChecked property, 129
described, 59
JavaScript syntax, 38
name property, 138
onClick event handler, 154
syntax for interactive forms, 22–23
type property, 146
value property, 147
checked property, 128
clearTimeout() method, 97
See also setTimeout() method
click() method, 92, 97–98
clickable maps, 31–32, 180
client
See Web clients
close (document) method, 98
close (window) method, 98
colon (:), separating hostname and
color port, 133
colors
type property, 146
value property, 147
vlinkColor property, 148
in Web pages, 164
comma (,) operator, 50
comment characters, special, 177–178
comments
syntax, 18
types of, 42
comparison operators, 48
complete property, 128
conditional expressions, 43, 45
conditional operators, 50
confirm() method, 98–99, 181–182
conforming to a pattern, field-level
validation, 186–187
consistency editing, 184
content providers, 198
continue keyword, 44
cookie property, 129
cookies, 178–179, 198
cost() method, 99
See also sin() method
currency
See money fields
custom colors, 191
custom functions, 88–90
custom messages, in status bar, 159
custom properties
See prototype property
domain, 198
data types, 37, 73, 141
See also names of specific data types
Date utility objects
described, 75–76
JavaScript syntax, 39
parse() method, 110
toGMTString() method, 120
toLocaleString() method, 121
days, specifying, 102, 113–114
debugging objects, 172
defaultChecked property, 129
defaultSelected property, 129
defaultStatus property, 129, 144
defaultValue property, 130
description property, 130
dialog boxes, invoked by onFocus event
handler, 156
displayBanner() function, 171
displayFlag() function, 169
division (/), JavaScript assignment
operator, 47
division (/), JavaScript mathematical
operator, 49
document objects
type property, 146
value property, 147
floating-point numbers, 85–86
floor() method, 100
focus, 198
focus() method, 100
tFontColor() method, 101,
tFontColor() method, 101, 164
fonts
color, 101
fixed-pitch, 99–100
italics, 106
size, 95, 101
sizefont() method, 101
for...in loops, 46
for loops, 43, 45–46
foreground (text) color, 11
form object, 184

<FORM> tag
ACTION attribute, 34
defining, 12–13
forms
See also interactive forms
defining with HTML, 12–13
described, 61
elements, 15–21
elements array, 130–131
encoding property, 131
JavaScript syntax, 38
method property, 137
onReset event handler, 160
onSubmit event handler, 161
reset() method, 112
submit() method, 119
target property, 145
forms array, 132
forward() method, 101–102
frame objects
clearTimeout() method, 97
creating, 18–19
described, 61–62
frames array, 132
JavaScript syntax, 39
name property, 138
onBlur event handler, 151–152
onFocus event handler, 155–157
onLoad event handler, 157–158
onUnload event handler, 161–162
parent property, 139
self property, 143
setTimeout() method, 115–116

<FRAME> tag, 18–19
frames array, 132
<FRAMESET> tag, 18–19
Function utility objects
arguments property, 127
described, 77
JavaScript syntax, 39
minus sign (-), in assignment operators, 47
minutes, specifying, 103, 114
modulus (%) JavaScript assignment operator, 47, 48
JavaScript mathematical operator, 49
money fields, formatting, 173-175
months, specifying, 102, 103, 113, 114
mouse clicks
See click() method
mouse pointer, 159
moving images, in Web pages, 165
multicolored bubbles, 30-31
multimedia, in Web pages, 164-165
MULTIPLE attribute, for <SELECT> element, 25
multiplication (*), JavaScript assignment operator, 47
multiplication (*), JavaScript mathematical operator, 49

N
NAME attribute, for radio buttons, 24
name property, 126, 138-139
natural logarithms, 108, 136
navigator objects
appCodeName property, 127
appName property, 127
appVersion property, 127
described, 66
describeEnabled() method, 106
JavaScript syntax, 39
mimeType property, 138
plugins array, 140
userAgent property, 147
navigator properties, 181
nested objects, 171, 172-173
nested quotes, 46-41
Netscapenavigator
accessing embedded plug-ins, 168
accessing images, 16
accessing source code, 177
ActiveX compatibility, 15, 167
calculating values for HTML tags, 183
clickable maps, 180
interacting with applets, 167
interacting with JavaScript, 167
JavaScript security issues, 36
JavaScript support, 3
LiveConnect technology, 199
LiveWire tool, 199
loading files, 179
onSelect event handler, 160
plug-in objects, 188-189
plug-ins, 32
reset() method, 112
resetting functions assigned to event handlers, 150
SRC attribute, 177
stopping onClick events, 153-154
viewing source code, 180
Netscape PowerStart, 178
new operator, creating objects, 89-90, 169
newline character
See writeln() method
not (!), JavaScript logical operator, 49
not equal to (!=), JavaScript comparison operator, 48
null data type, syntax, 37
data type, syntax, 37
numeric, field-level validation, 185-186
numeric computations
See also integers; values
cos() method, 99
eval() function, 84-85
floor() method, 100
LN2 property, 136
LN10 property, 136
LOG2E property, 137
max() method, 108
min() method, 108
random() method, 111
setInterval() method, 115-116
sin() method, 116
sort() method, 117
sqrt() method, 118
SORT1 property, 144
SORT2 property, 144
tan() method, 120

O
object model, 36-40
<OBJECT> tag, embedding ActiveX component, 15-16
objects
See also HTML (HyperText Markup Language): names of specific objects
blur() method and, 95-96
bypassing, 51
creating, 169-170
custom functions and, 89-90
debugging, 172
described, 56, 200
focus() (Interlaced), 100
length property, 135-136
NAME attribute, 138-139
nested, 171-173
properties, 126
prototype property, 141
reserved words, 189-190
type property, 146
value property, 147
onAbort event handler, 150
onBlur event handler, 151-152
onChange event handler, 152-153
onClick event handler, 153-155
onError event handler, 155
onFocus event handler, 155-157, 198
onLoad event handler, 157-158, 171
onMouseOver event handler, 51, 158-159
onMouseOver event handler, 51, 159
onReset event handler, 160
onSelect event handler, 160
onSubmit event handler, 161, 184
onUnload event handler, 161-162
open (document) method, 108-109, 139
open (window) method, 109-110, 139
opener property, 139
operators, 47-52
Option utility objects
defaultSelected property, 129
described, 77-78
index property, 134
JavaScript syntax, 40
name property, 138
selected property, 143
text property, 145
value property, 147
<option> tag, repeating, 26
options array, 139, 143, 147
or (!), JavaScript logical operator, 49, 50
output stream, 108-109

P
pairs, in JavaScript, 41
parameters, for custom JavaScript functions, 86-89
parent property, 139-140
parentheses
omitting in event handlers, 150
precedence order for JavaScript operators, 49, 50
parse() method, 110
parseInt() function, 85-86, 185
parseInt() function, 86-87, 185
password objects
defaultValue property, 130
defining for interactive forms, 24
described, 66-67
JavaScript syntax, 38
name property, 138
onClick event handler, 154
type property, 146
value property, 147
path name, image source, 165
pathname property, 140
PI property, 140
pictures, in Web pages, 165
See also image objects
platform dependence, 121
plug-in objects
accessing with JavaScript, 32-33
for animation, 165
described, 67
description property, 130
documentation, 168, 169
embedding in HTML files, 168
Envoy document management, 168
filename property, 132
JavaScript syntax, 39
name property, 138
specific values, 169
plugins array, 140
plus sign (+), in assignment operators, 47
polymorphism, 200
popup messages, 181-182
port address, 133
port property, 140-141
port value, default, 141
pow() method, 110
prompt() method, 110, 118-182
properties, 126, 138, 146, 147, 200
See also names of specific properties;
objects
protocol property, 141
prototype property, 141
public keyword, 200
punctuation in JavaScript, 40

Q
question mark (?), in search values, 142
radio buttons
checked property, 128
click() method, 97
defaultChecked property, 129
defining for interactive forms, 24-25
described, 67-68
implementation, 139
JavaScript syntax, 38
name property, 138
onClick event handler, 154
type property, 146
value property, 147
radix argument, 86-87
random() method, 111
referrer property, 142
relational operators, 49
reload() method, 111
reset() method, 112
reset objects
described, 68
for interactive forms, 25
JavaScript syntax, 38
name property, 138
reset objects (continued)
onClick event handler, 154
type property, 146
value property, 147
result strings, 46
reverse() method, 112
RGB triplet values, 148, 191
round() method, 112

S

S<SCRIPT> tag
accessing source code, 177
defining, 13-14
scripts, 200
scroll() method, 113
scrolling text, displaying, 170-171
search property, 142
seconds, specifying, 103-104, 115
select() method, 113
select objects
described, 68-69
JavaScript syntax, 38
name property, 138
onBlur event handler, 151-152
onChange event handler, 152-153
onFocus event handler, 155-157
options array, 139
selectedIndex property, 143
type property, 146
<select> element, defining interactive forms, 25-26
selected property, 143
selectedIndex property, 143
self property, 143
See also window property
servers
See Web servers
setDate() method, 113-114
setHours() method, 114
setMinutes() method, 114
setMonth() method, 114
setSeconds() method, 115
setTime() method, 115
setTimeOut() method, 115-116
setYear() method, 116
Shockwave for Director, Macromedia, 165
showHelp() function, 160
sin() method, 112
single quote (‘), surrounding string values, 40
slash-asterisk (/\*)
HTML comments, 18
JavaScript comments, 42
small() method, 117
sort() method, 117
sound, in Web pages, 165
source code order, 131, 132, 135-136
source files, 177, 180, 201
special characters, encoding with escape() function, 84
special comments characters, 177-178
spelling, 41-42, 82
split() method, 118
sqrt() method, 118
sQRT1_2 property, 144
sQRT2 property, 144
square brackets ([), with conditional expressions, 43, 88
SRC attribute, 177
src property, 144
Start Over button, Netscape
PowerStart, 179
state, 201
Sel also objects
status bar, setting value for, 159
status property, 144
strike() method, 118
string arguments
with escape() function, 84
with methods, 92
with parseInt() function, 86
with parseFloat() function, 87
with unescape() function, 87
with unescape() function, 87
string data type, syntax, 37, 79
String objects
big() method, 95
described, 79-80, 201
fontColor() method, 164
indexOf() method, 105-106
length property, 117
split() method, 118
strike() method, 118
sub() method, 119
substring() method, 119-120
sup() method, 120
toLowerCase() method, 121
toString() method, 121-122
toUpperCase() method, 122
String operators, 52
sub() method, 119
submit buttons
click() method, 97
described, 69
in interactive forms, 27, 34
JavaScript syntax, 38
name property, 138
onClick event handler, 155
type property, 146
value property, 147
substring() method, 119
substring() method, 119-120
substrings, 201
subtraction (-), JavaScript assignment operator, 47

subtraction (-), JavaScript mathematical operator, 49
suffixes property, 144
sup() method, 120
syntax
ActiveX component, 15-16
anchors, 19-20
arrays, 74
buttons for interactive forms, 22
check boxes for interactive forms, 22-23
custom JavaScript functions, 88-90
Date utility objects, 75
described, 201
fileUpload element for interactive forms, 23
frames, 19
Function utility objects, 77
hidden element in interactive forms, 23
HTML comments, 18
HTML elements, 5-7
hypertext links, 20
<img> tag, 17
<input> element, 21-25
interactive forms, 13
Java applets, 30-31
JavaScript event handlers
Option utility objects, 78
password object in interactive forms, 24
plug-ins, 32-33
radio buttons in interactive forms, 24-25
reset object in interactive forms, 25
<select> element, 25-26
string objects, 37, 79
submit buttons, 27, 34
text elements, 27-28
<textarea> element, 28-29
variable scope, 54

T
tags, in HTML, 6-7, 201
See also HTML (HyperText Markup Language)
tan() method, 120
target property, 145
template, 201
See also HTML (HyperText Markup Language)
texticographic order, 117
TEXT attribute, fColor property, 131
text color, 164
text files, saving JavaScript as, 182-183
text objects
defaultValue property, 130
described, 69-70
in interactive forms, 27-28
JavaScript syntax, 38

name property, 138
onBlur event handler, 151-152
onChange event handler, 152-153
onFocus event handler, 155-157
onSelect event handler, 160
scrolling, 170-171
select() method, 113
type property, 146
value property, 147
text property, 145
textarea objects
defaultValue property, 130
described, 70
JavaScript syntax, 39
name property, 138
onBlur event handler, 151-152
onChange event handler, 152-153
onFocus event handler, 155-157
onSelect event handler, 160
select() method, 113
type property, 146
value property, 147
<TEXTAREA> element, defining interactive forms, 25, 28-29
this keyword, 90, 201
time, specifying, 104, 115
time zone, specifying, 104
title property, 145
<TITLE> tag, 34
toGMTString() method, 120
toLocaleString() method, 121
toLowerCase() method, 121
toUpperCase() method, 122
toString() method, 121-122
toUpperCase() method, 122
transaction, 201
Tumbleweed Software, Envoy document management plug-in, 168
TYPE attribute, for <INPUT> tag, 28
type property, 146
typeof operator, 50-51

U

unary operators
examples, 52
syntax, 49
unescape() function, 87
Universal Coordinated Time, 122
update expressions, 45
URL (Uniform Resource Locator), 3, 201
user input, validating, 183
userAgent property, 147
user-generated events, 150, 180
UTC() method, 122
utility objects, 39-40
U
validating user input, 183–187
value property, 126, 147
values
See also integers; numeric computations; properties
ALINK attribute, 126
for angles, 94
assigning to variables, 53
calculating for HTML tags, 183
of colors, 191–195
for integers, 96
isNaN() function and, 85
for METHOD attribute of forms, 137
of ports, 141
resetting for event handlers, 150
returned by functions, 83
round() method, 112
for searches, 142
sort() method, 117
status property, 144
user-agent header, 147
VLINK attribute, 148
variables, 53–54, 189–190
verifyExistence() function, 152
VLINK (followed link) color, 164
vlinkColor property, 148
void operator, 51
vspace property, 148

W
Web
application development, 2–3
described, 202
plug-ins, 33
protocols, 2
search engines, 142
Web browsers
hiding source code, 176–177
script compatibility, 181
Web clients, 2–3, 198, 202
Web pages
adding multimedia to, 164–165
bookkeeping, 34
clickable maps, 31–32
configuring, 33–34
content, 198
creating links
to other pages, 187–188
within a single page, 188
creating with HTML, 10–14
displaying money values, 173
embedding ActiveX in, 167
embedding images in, 16–18
embedding Java applets in, 29–31, 167–168
interactive, 149
JavaScript template for, 176
plain-vanilla version, 181
plug-ins for sale, 33
referencing .js files in, 177
scrolling text in, 170–171
title, 34
Web servers, 2, 202
Web sites
ActiveX plug-in for Netscape Navigator, 167
described, 202
Envoy document management plug-in, 168
for JavaScript security issues, 36
JavaScript-related, 142
Netscape PowerStart, 179
while loops, 43, 46–47
width property, 148
window objects
clearTimeout() method, 97
confirm() method, 98–99
defaultStatus property, 129, 144
described, 70–71
displaying pop-up dialog box, 93
frames array, 132
JavaScript syntax, 38
methods and, 92
name property, 138
onError event handler, 155
onFocus event handler, 155–157
onLoad event handler, 157–158
onUnload event handler, 161–162
opener property, 139
parent property, 139–140
prompt() method, 110
scroll() method, 113
self property, 143
setTimeout() method, 115–116
status property, 144
top property, 146
window property, 148
with keyword, 90
wizards, 202
World Wide Web
See Web
write() method, 122–123
writeln() method, 123
WYSIWYG (What You See Is What You Get), 202

Y
year, specifying, 104–105, 116
Includes Complete JavaScript Syntax for Netscape Navigator and Internet Explorer!

Fun, Fast & Cheap!

A Quick Reference for the Rest of Us!

Everybody's creating Web pages for fun and profit—and now you can too, with JavaScript, an easy-to-use scripting language! Keep JavaScript® For Dummies® Quick Reference handy for tips on everything you need to know about building interactive Web pages. Whether you're a seasoned HTML pro or you're just beginning your Internet adventure, this complete, up-to-the-minute JavaScript language reference will save you time and help you create attractive, compelling Web pages.

Find what you need quickly with our For Dummies® Quick References! The design features tasks and commands in alphabetical order, cut-and-paste, step-by-step instructions, and easy-to-follow advice. "Get in and get out" quickly and get the information you really need without reading lots of extra material!

Look for IDG Books Worldwide's JavaScript™ For Dummies® for more information on creating your own Web page.

For Dummies Quick References and For Dummies books are available on all your favorite or not-so-favorite hardware and software products. Look for them wherever computer books are sold!

Let These Icons Guide You!

- Tip: Points out time-saving shortcuts and handy tricks
- Warning: Warns you of common mistakes and tells you how to avoid them
- Example: Alerts you to real-life, working examples you can use in your own Web pages

JavaScript is a trademark of Sun Microsystems, Inc.
The IDG Books Worldwide logo is a registered trademark under exclusive license to IDG Books Worldwide, Inc., from International Data Group, Inc. The ...For Dummies logo is a trademark, and Dummies Man, For Dummies, A Quick Reference for the Rest of Us, Your Fingertip Companion, Fun, Fast & Cheap, and Dummies Press are registered trademarks of IDG Books Worldwide, Inc.

Printed in the USA.