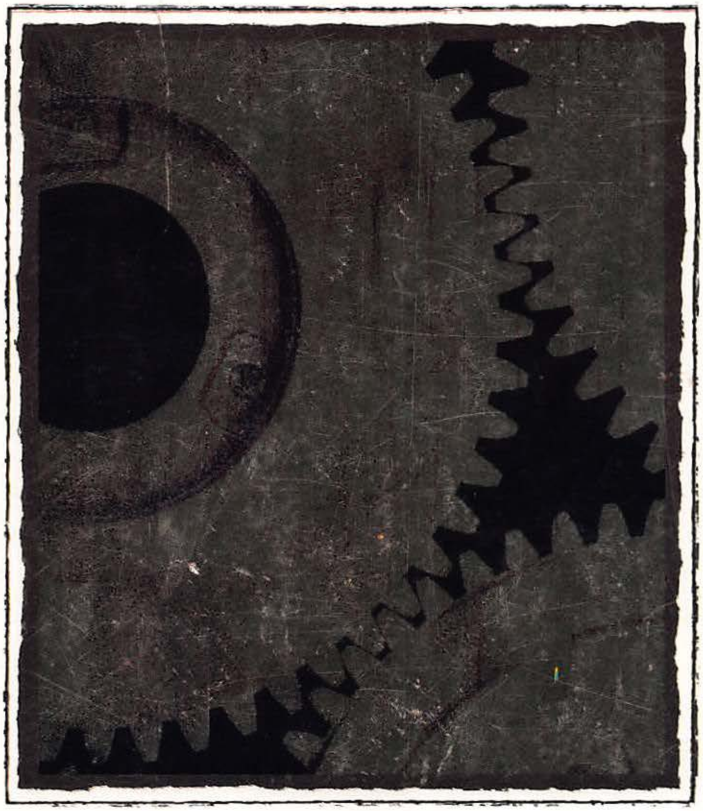
Hayden Books



CD-ROM INCLUDES COOL JAVASCRIPTS

Add Flexibility and Interactivity to Your Web Pages with JavaScript

JAVASCRIPT FOR MACINTOSH®

Matt Shobe and Tim Ritchey

JavaScript for Macintosh®

Matt Shobe

Tim Ritchey

Hayden Books

Hayden Books

Publisher
Lyn Blake

Marketing Manager
Nancy Price

Publishing Manager
Laurie Petrycki

Managing Editor
Lisa Wilson

Acquisitions Editor
Brian Gill

Development Editor
Steve Mulder

Copy/Production Editor
Bront Davis

Technical Editor
Mason Hale

Publishing Coordinator
Rosemary Lewis

Cover Designer
Karen Ruggles

Book Designer
Sandra Schroeder

Manufacturing Coordinator
Brook Farling

Production Team Supervisor
Laurie Casey

Production Team
Heather Butler
Angela Calvert
Dan Caparo
Kim Cofer
Terrie Deemer
Tricia Flodder
Aleata Howard
Joe Millay
Erika Millen
Christine Tyner
Karen Walsh

Indexer
Tom Dinse

JavaScript for Macintosh®

Library of Congress Catalog Number: 96-75189

ISBN: 1-56830-278-9

Printed in the United States of America 1 2 3 4 5 6 7 8 9 0

Warning and Disclaimer

This book is sold as is, without warranty of any kind, either express or implied. While every precaution has been taken in the preparation of this book, the authors and Hayden Books assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information or instructions contained herein. It is further stated that the publisher and authors are not responsible for any damage or loss to your data or your equipment that results directly or indirectly from your use of this book.

Trademark Acknowledgments

Hayden Books

The staff of Hayden Books is committed to bringing you the best computer books. What our readers think of Hayden is important to our ability to serve our customers. If you have any comments, no matter how great or how small, we'd appreciate your taking the time to send us a note.

You can reach Hayden Books at the following:

Hayden Books
201 West 103rd Street
Indianapolis, IN 46290
(800) 428-5331 voice
(800) 448-3804 fax

Email addresses:

America Online:	Hayden Bks
Internet:	hayden@hayden.com

Visit the Hayden Books Web site at http://www.hayden.com

Contents at a Glance

Table of Contents

INTRODUCTION

With the release of Netscape Navigator 2.0 for Macintosh, Steve Jobs must be pleased simply because Netscape 2.0 enables unrealized new potential for "insanely great" Web pages. Netscape's latest browser, fully caffeinated with the new JavaScript language, enables first-time Web page designers, with little programming experience, to create Web pages that can "listen" and respond to user actions quickly and intuitively. With a little JavaScript code mixed in to the HTML documents you're already writing today, your Web pages can become more interactive, professional, and *fun to use* than you ever thought possible. Of course, you want to get your site up-to-speed with JavaScript as soon as possible. This book takes you through this exciting and powerful new language and gives you the ability to create dynamic and interactive Web pages immediately, using lots of examples. JavaScript is based upon Java, a full-blown, object-oriented development environment that is very powerful and naturally takes a long time to learn effectively. The fact that Java is complex and intricate does not mean JavaScript is too; in fact, anyone who can program in BASIC or infer that the code snippet

```
document.write ("Hey Mom! Send money!");
```

will simply print a line on the screen has a great shot at mastering JavaScript. On the flip side, that JavaScript is a simplified subset of Java does *not* mean that you don't have much power. As you will see, JavaScript offers many flexible ways to make your Web pages more like real Macintosh applications that accept user input and respond *quickly*.

The Purpose of This Book

This book was written to lead you through the JavaScript programming language and to provide you with the skills you need to develop scripts that add vitality and interactivity to your Web pages.

You will learn how to program JavaScript into your pages so that they can respond to users and their "client-side" computing environment. By sending all your scripts to the client machine (the computer at the other end of a modem or network from your Web server), as JavaScript does, you can do away with bothersome and often arcane CGI scripting. CGI scripting too often means scrupulously maintaining not only your Web documents, but also the CGI program files and directory structure on the server machine. With JavaScript, you incorporate code in an HTML document, which binds document and script as a single point of maintenance. If you want to change something in a script-enabled page, you simply load up your favorite text editor and change it just like any other HTML or plain text document. And even after you have learned the intricacies of JavaScript, you can apply your knowledge to the more complex and powerful world of Java applets, because JavaScript's syntax and concepts are established by its big brother, Java.

If you have already done any programming in JavaScript or Java, then this book can provide you with a single handy reference to both of these languages. The special reference section in the back of the book should prove especially helpful when you are pounding out that long script and can't remember the name of a method or the arguments the history object takes.

If you are an experienced programmer who wants to learn more about JavaScript, this book will guide you through the language and point out comparisons to and differences from other popular languages such as C and C++. Because Java and JavaScript are loosely based on the C++ language, you should have no problems getting up-to-speed quickly and moving through both sections on Java and JavaScript, enabling you to start writing programs as soon as possible.

The Organization of This Book

The first chapter introduces you to some of the cool stuff you can do with JavaScript. Chapters 2 through 8 present the JavaScript language and the Netscape Navigator objects that JavaScript can interact with.

2. Introduction to the Java Family
3. JavaScript Fundamentals
4. Control Flow and Functions in JavaScript
5. Using and Creating Objects in JavaScript
6. JavaScript and Built-In Objects
7. Netscape Navigator Objects: The Document Object
8. The Forms, Window, History, and Location Objects

The appendices provide several references for the JavaScript language and objects, as well as a collection of fine examples of JavaScript in action around the Net.

Conventions Used in This Book

Before you begin to read, you should be aware of several different types of print that are used.

- ☐ `Print that looks like this` is program code.
- ☐ *`Print that uses italics in programming code`* indicates the word is a variable—the actual word varies according to you and your situation.
- ☐ *Italics* are also used to indicate ***new words*** that are followed by a definition.

Icons also are used to illustrate notes and warnings. The author made an effort to set these aside for your added information.

NOTE Notes include extra information or useful tips that complement the discussion.

WARNING A warning cautions you to be aware of certain tricky situations that can arise during a particular step in a process. Make sure you read these carefully.

With these special formatting conventions in mind, you should find this book easier to read and remember.

Before You Begin

For the JavaScript sections of this book, you will need only the Netscape Navigator 2.0 browser and a plain text editor, such as SimpleText or BBEdit. You can check out Netscape's site at http://home.netscape.com for downloading instructions.

JavaScript enables you to immediately change the look and feel of your Web pages. It also serves as a worthy introduction to more sophisticated and powerful capabilities of Java, which might drag you even deeper into this nasty, addictive business of Internet content programming. Have no fear. What you learn here will only entertain, improve the interactive experience for your Web site's users, and make your trusty Mac a powerful Web content development workstation. Good luck and smooth programming.

Matthew Shobe, Seattle, Washington

Timothy Ritchey, Cambridge, England

CHAPTER 1

Cool Stuff You Can Do with JavaScript

This chapter could be considered the "Quick Start" section of the book, because we want to show you right away some of the potential JavaScript has for improving Netscape users' everyday experience with Web pages they access—especially yours. Whether you bought this book with the goal of improving your own personal home page or revitalizing your company's entire Web site, you should know that JavaScript will require you to think about the capabilities of these pages differently. No longer is their content pretty, yet static at best. Items on the page, such as hyperlinks and form elements (for example, buttons, text areas, and list boxes), now can respond to user actions directly and with great speed, because the programming that drives them is downloaded to the user's computer. Just moving the mouse pointer over a link, for example, could trigger a JavaScript-enabled page to display a brief summary of that link's contents in the Netscape status bar at the bottom of the window, instead of an inscrutable URL. Also, clicking a button could cause a particular image on the page to be updated, changed, or affected by values the user has entered.

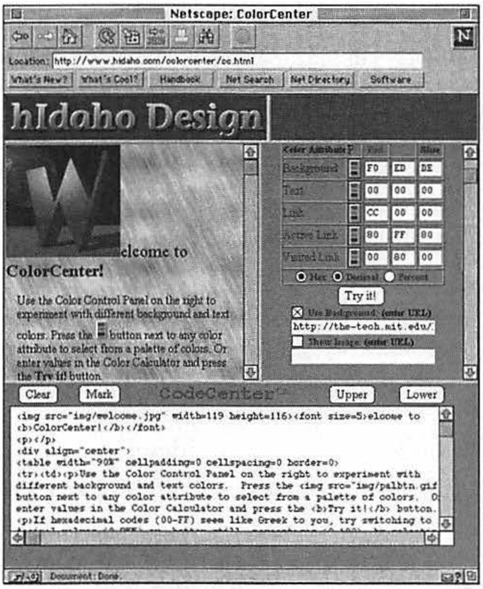
The following sections describe potential application areas and provide examples from the Web or ones we have written. Don't be concerned if you don't know a thing about JavaScript syntax or programming in general at this point; we're simply taking the "cooking show approach" and previewing the finished dishes without a list of ingredients or instructions on how to combine them—yet. Sit back, fire up Netscape 2.0, and try these links to see how they act on your Macintosh.

Stand-Alone Applications

JavaScript sites in this category attempt to transcend the invisible boundary that separates "read-only" Web page content from an actual application that supports a particular task or user goal. Before JavaScript, you could read a Web page that explained how to create colored backgrounds, text, and links; with scripting, the same page can now dynamically help you pick colors, test background patterns, and copy the results for your own later use. The potential for JavaScript changing the way we view the Web seems greatest here—imagine applications that can apply current stock prices to the value of a portfolio you've predefined, or host a live auction of secondhand computer equipment in which you can instantly receive bids or place your own goods on the block.

ColorCenter™

http://www.hidaho.com/colorcenter/cc.html



Location: http://www.hidaho.com/colorcenter/cc.html

Use the Color Control Panel on the right to experiment with different background and text colors. Press the ▤ button next to any color attribute to select from a palette of colors. Or enter values in the Color Calculator and press the Try it! button.

Color Attribute		Red		Blue
Background	▤	F0	ED	DE
Text	▤	00	00	00
Link	▤	CC	00	00
Active Link	▤	80	FF	80
Visited Link	▤	00	80	00

☒ Use Background: (enter URL)
http://the-tech.mit.edu/
□ Show Image: (enter URL)

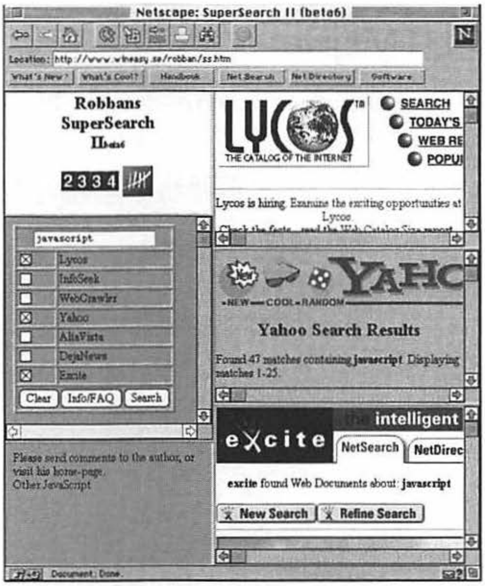
```
<img src="img/welcome.jpg" width=119 height=116><font size=5>elcome to
<b>ColorCenter!</b></font>
<p></p>
<div align="center">
<table width="90%" cellpadding=0 cellspacing=0 border=0>
<tr><td><p>Use the Color Control Panel on the right to experiment with
different background and text colors.  Press the <img src="img/palbtn.gif
button next to any color attribute to select from a palette of colors.  O
enter values in the Color Calculator and press the <b>Try it!</b> button.
<p>If hexadecimal codes (00-FF) seem like Greek to you, try switching to
```

Figure 1.1 *The ColorCenter™, a JavaScript-enhanced Web page.*

This is the JavaScript color-choosing example we oh-so-subtly mentioned in the previous paragraph. As shown in Figure 1.1, ColorCenter enables its users to preview HTML link and background color choices directly in the frame shown in the upper left-hand corner of the browser window. Using the CodeCenter feature, you can even cut and paste the HTML of a page you're currently working on and see how it might look with several color and background pattern treatments. Although the interface is a bit complicated, the usefulness of this application for previewing color choices should be clear to anyone who has ever scratched his head, wondering what the hexadecimal code for a certain shade of chartreuse might be.

SuperSearch

http://www.wineasy.se/robban/ss.htm



Netscape: SuperSearch II (beta6)

Location: http://www.wineasy.se/robban/ss.htm

Robbans SuperSearch IIbeta6

javascript

☒ Lycos
□ InfoSeek
□ WebCrawler
☒ Yahoo
□ AltaVista
□ DejaNews
☒ Excite

Please send comments to the author, or visit his home-page.
Other JavaScript

Lycos is hiring. Examine the exciting opportunities at Lycos.

Check the facts... read the Web Catalog Size report

Yahoo Search Results

Found 43 matches containing **javascript**. Displaying matches 1-25.

excite found Web Documents about: **javascript**

Document: Done.

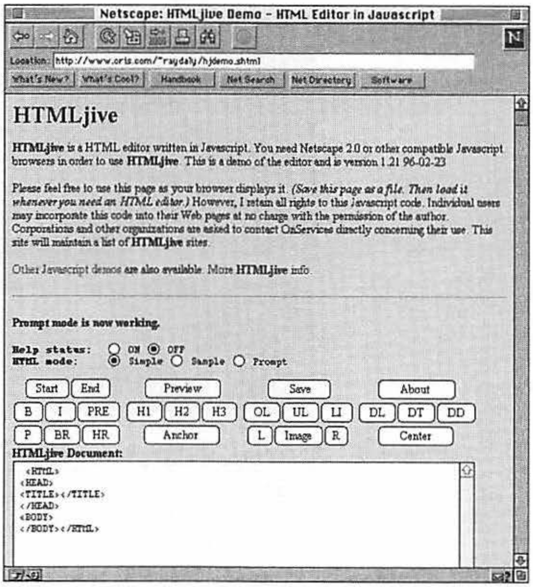
Figure 1.2 *The SuperSearch site combines seven of the Net's top search sites.*

Even if you've browsed the Web only once, odds are you've visited at least one site that offers a comprehensive, searchable index of millions of existing Web sites. The only trouble is, each one uses somewhat different methods of cataloging Internet content, and so the same search can often produce different results at different sites. Thus, to really know what's available out there on the topic of, say, JavaScript, you need to visit each site, type your keywords, and start exploring the results.

Robban's SuperSearch is an excellent solution for reducing the amount of travel you need to make between Lycos, Yahoo, Excite, and others. We consider it a JavaScript application because it acts as a middleman between you and several search engines. Type your search keywords, select which search sites you want to scour, and watch the results from each search engine appear in frames on the right-hand side.

HTMLjive

http://www.cris.com/~raydaly/hjdemo.shtml



Netscape: HTMLjive Demo - HTML Editor in Javascript
Location: http://www.cris.com/~raydaly/hjdemo.shtml
What's New?
What's Cool?
Handbook
Net Search
Net Directory
Software

HTMLjive

HTMLjive is a HTML editor written in Javascript. You need Netscape 2.0 or other compatible Javascript browsers in order to use **HTMLjive**. This is a demo of the editor and is version 1.21 96-02-23

Please feel free to use this page as your browser displays it. *(Save this page as a file. Then load it whenever you need an HTML editor.)* However, I retain all rights to this Javascript code. Individual users may incorporate this code into their Web pages at no charge with the permission of the author. Corporations and other organizations are asked to contact OnServices directly concerning their use. This site will maintain a list of **HTMLjive** sites.

Other Javascript demos are also available. More **HTMLjive** info.

Prompt mode is now working.

Help status: ○ ON ● OFF
HTML mode: ● Simple ○ Sample ○ Prompt

HTMLjive Document:

```
 <HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
</BODY></HTML>
```

Figure 1.3 *HTMLjive, an HTML editor written in JavaScript.*

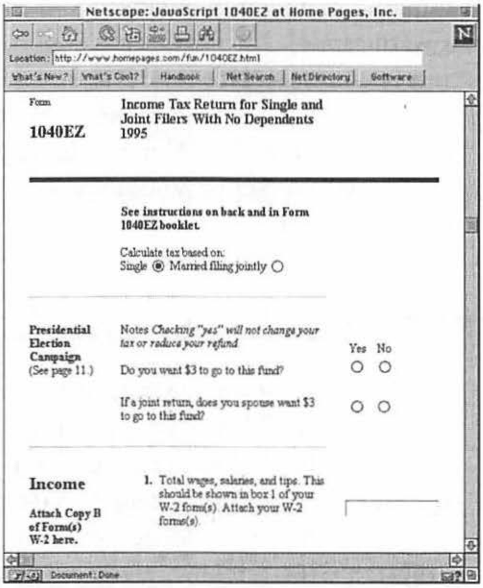
HTMLjive is a page that can truly stand on its own. This site offers an HTML editor written entirely in JavaScript; although it pales in comparison to full-featured editors, it is novel simply because we have here Web content that can create *more* Web content. There's something hideously organic about this. The author enables you to save the source file for this page on your Macintosh; if you do this, and open it with Netscape 2.0, you have an HTML editor that works much like a stand-alone application—no network connection needed to run it. HTMLjive only hints at the possibilities that sophisticated JavaScript (and, more likely, Java) can deliver directly to your computer.

Worksheets, Lookups, and Calculators

Using JavaScript's computational capabilities on the end user's machine only makes sense—if you fill out an HTML 1040 income tax form, why should a remote server use a CGI script to compute your tax when you've got a perfectly good Macintosh CPU sitting right in front of you? Moreover, do you really *want* to send your Adjusted Gross Income sailing across the Infobahn? Sites in this category offer unique worksheets and models for solving mathematical problems, both commonplace and arcane. Using JavaScript to deliver customized calculators, what-if analysis worksheets, and simple database lookup tables across a local network or the entire Internet might be one of the easiest ways to learn the language and still provide a dynamic, unique experience for your Web sites' users.

1040EZ Income Tax Return

http://www.homepages.com/fun/1040EZ.html


Netscape: JavaScript 1040EZ at Home Pages, Inc.
Location: http://www.homepages.com/fun/1040EZ.html
What's New?
What's Cool?
Handbook
Net Search
Net Directory
Software
Document: Done

Form **1040EZ**

Income Tax Return for Single and Joint Filers With No Dependents 1995

See instructions on back and in Form 1040EZ booklet.

Calculate tax based on:
Single ● Married filing jointly ○

Presidential Election Campaign (See page 11.)

Notes *Checking "yes" will not change your tax or reduce your refund*

	Yes	No
Do you want $3 to go to this fund?	○	○
If a joint return, does you spouse want $3 to go to this fund?	○	○

Income

Attach Copy B of Form(s) W-2 here.

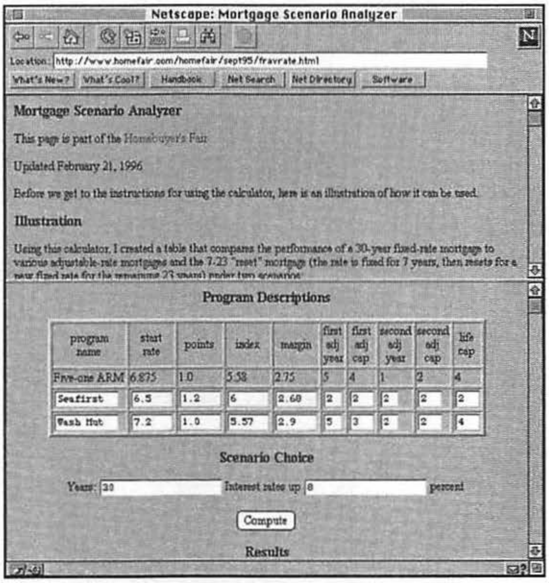
1. Total wages, salaries, and tips. This should be shown in box 1 of your W-2 form(s). Attach your W-2 forms(s).

Figure 1.4 *Uncle Sam, meet JavaScript: the 1040EZ Income Tax Return.*

Sure, the 1040EZ is a pretty easy return as IRS forms go, but because authors can immediately update their Web pages' content, wouldn't an entire battery of forms online, available for download, be the next logical step? Each form should be updated to also show the latest IRS rules, deductions, exemptions, and, well, loopholes? Hello, Intuit?

Mortgage Scenario Analyzer

http://www.homefair.com/homefair/sept95/fravrate.html



Location: http://www.homefair.com/homefair/sept95/fravrate.html

Mortgage Scenario Analyzer

This page is part of the Homebuyer's Fair

Updated February 21, 1996

Before we get to the instructions for using the calculator, here is an illustration of how it can be used.

Illustration

Using this calculator, I created a table that compares the performance of a 30-year fixed-rate mortgage to various adjustable-rate mortgages and the 7-23 "reset" mortgage (the rate is fixed for 7 years, then resets for a new fixed rate for the remaining 23 years) under two scenarios:

Program Descriptions

program name	start rate	points	index	margin	first adj year	first adj cap	second adj year	second adj cap	life cap
Five-one ARM	6.875	1.0	5.58	2.75	5	4	1	2	4
Seafirst	6.5	1.2	6	2.60	2	2	2	2	2
Wash Mut	7.2	1.0	5.57	2.9	5	3	2	2	4

Scenario Choice

Years: 30 Interest rates up 8 percent

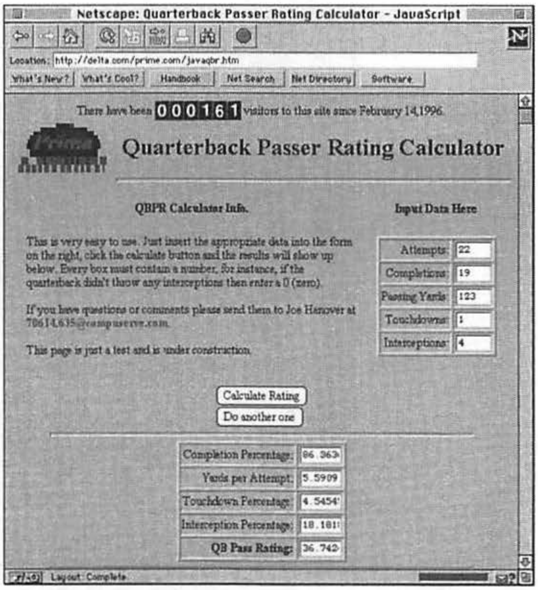
Results

Figure 1.5 *Buying a home? Try the Scenario Analyzer and wow your banker.*

If you're a homeowner or planning to become one, one of the trickiest aspects of the entire purchase process can be choosing the right mortgage. Do I choose a fixed, adjustable, or 7–23 reset rate? This Web page offers a sophisticated yet simple blade for cutting the homebuyer's Gordian Financing Knot. Access this site, plug in some numbers, and run some what-if scenarios. It sure beats correctly entering the formulas yourself in Excel.

Quarterback Passing Rater

http://delta.com/prime.com/javaqbr.htm



Netscape: Quarterback Passer Rating Calculator - JavaScript
Location: http://delta.com/prime.com/javaqbr.htm
What's New?
What's Cool?
Handbook
Net Search
Net Directory
Software
Layout: Complete

There have been 000161 visitors to this site since February 14,1996.

Quarterback Passer Rating Calculator

QBPR Calculator Info.

This is very easy to use. Just insert the appropriate data into the form on the right, click the calculate button and the results will show up below. Every box must contain a number, for instance, if the quarterback didn't throw any interceptions then enter a 0 (zero).

If you have questions or comments please send them to Joe Hanover at 70614.635@compuserve.com.

This page is just a test and is under construction.

Input Data Here

Attempts:	22
Completions:	19
Passing Yards:	123
Touchdowns:	1
Interceptions:	4

Completion Percentage:	86.363
Yards per Attempt:	5.5909
Touchdown Percentage:	4.5454
Interception Percentage:	18.181
QB Pass Rating:	36.742

Figure 1.6 *Armchair quarterbacks, unite! Rate Troy, Steve, Warren, or Brett with stats from the Monday morning sports page.*

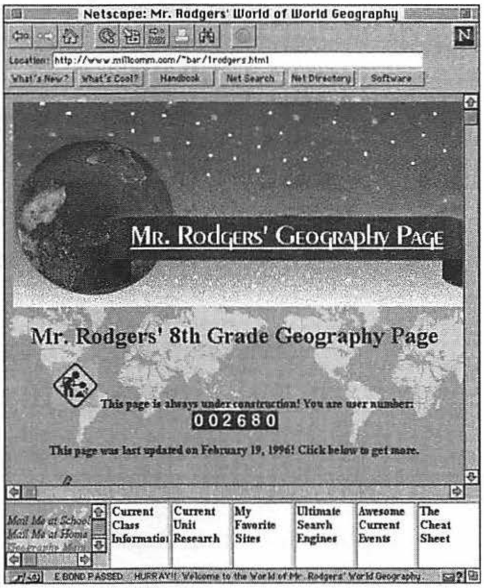
It was only a matter of time before sports fans realized the Net is a great place to both transmit and analyze their favorite athletes' statistics. With JavaScript, making and displaying calculations are relatively straightforward tasks. An interesting extension of this site might involve buttons that download the necessary passing statistics from ESPNet SportsZone for all the NFL quarterbacks who played on a given Sunday.

Subtly JavaScript-Enhanced HTML

Ahh, the art of being discreet. If you can use JavaScript to quietly improve the appearance and performance of a typical HTML site, why not do just that? The sites in this category don't take forever to transmit, and look for the most part like ordinary Web pages (whatever those might be). But JavaScript is in the details, friends.

Mr. Rodgers' World of World Geography

http://www.millcomm.com/~bar/1rodgers.html



Netscape: Mr. Rodgers' World of World Geography

Location: http://www.millcomm.com/~bar/1rodgers.html

Mr. Rodgers' 8th Grade Geography Page

This page is always under construction! You are user number: 002680

This page was last updated on February 19, 1996! Click below to get more.

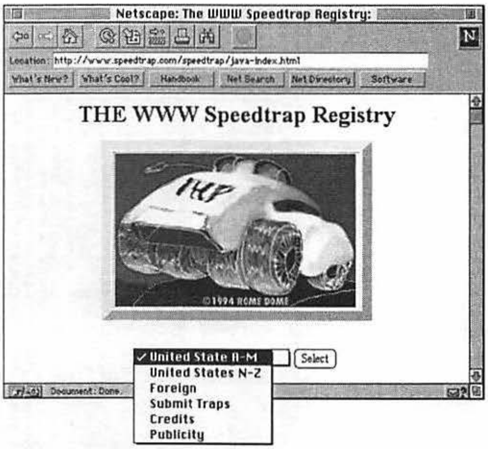
E BOND PASSED HURRAY!! Welcome to the World of Mr. Rodgers' World Geography.

Figure 1.7 *Is this the future of junior high education?*

Clearly, Mr. Rodgers wants the kids in his classroom to be well-informed, wired, and wise about the world. (Our apologies, but we have an alliteration quota to fill.) His home page offers everything you might need to keep up in his class as a student or look in on his methods as a parent. The subtle JavaScript at work here is the scrolling marquee in the status area of the Netscape window—see it down there on the bottom? Apparently, Mr. Rodgers is pleased that a recent school district bond levy passed. Scrolling marquees like this one are a neat way to draw the user's attention using motion and to highlight late-breaking information. Keep the marquee message short, though, or risk quickly losing users' attention.

The WWW Speedtrap Registry

http://www.speedtrap.com/speedtrap/java-index.html



Netscape: The WWW Speedtrap Registry:

Location: http://www.speedtrap.com/speedtrap/java-index.html

THE WWW Speedtrap Registry

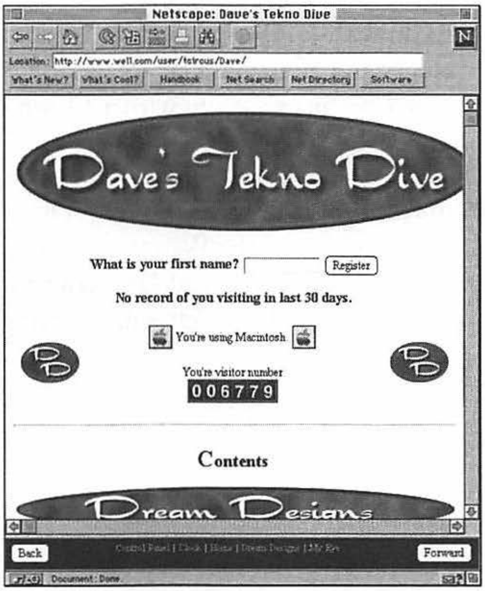
Document: Done.

Figure 1.8 *Keeping one step ahead of the law, state-by-state.*

The popular Speedtrap Registry helps leadfoots in 50 states avoid known radar traps, although, according to its author, "This page is not meant to be an effort to undercut the efforts of police to control motorists' speeds on dangerous roads." Regardless of your stance on its First Amendment rights, as a user you will be pleased by its simplified navigation structure, thanks to JavaScript. In the old version of the Registry, you had to choose your state from a list of 50 links on one page; that's a lot of vertical scrolling to reach Wyoming. The JavaScript-enhanced version simply subcategorizes them using the pop-up menu at the bottom center of the page, splitting the states A–M and N–Z. Forms for submitting new traps and other feedback also are accessible through this menu. Simply click Select and you're taken to the form of your choice.

Dave's Tekno Dive

http://www.well.com/user/tcircus/Dave/



Netscape: Dave's Tekno Dive

Location: http://www.well.com/user/tcircus/Dave/

What is your first name?

No record of you visiting in last 30 days.

You're using Macintosh

You're visitor number

006779

Document: Done.

Figure 1.9 *Dave's got ahold of your cookie. Ouch!*

Although there's not much subtlety in Dave's design aesthetic, his use of JavaScript to personalize your visit to his home page is something you should expect to see a lot more often. Using a Netscape feature called a *cookie*, Dave can tell whether you've visited his site before and potentially present customized content to you when you visit. Using his Control Panel, you can even set the background color for his page to your liking. You'll learn how to use the cookie in Chapter 7, "The Form, History, and Document Objects."

Summary

In this chapter you've been given a sampling of what's possible with JavaScript. Hopefully you noticed a pattern here: most of these sites' authors are mere mortals who've seen the potential of JavaScript and realized their vision in a Web site that works. Just like you, they were interested in what scripting could do to improve their content and more important, the users' experience while at their site. Keep the potential user in mind as you read through this book; you might think of something really technically challenging to do with JavaScript, such as 16 floating toolbars with flashing clocks and animated GIF images on each one, but what about the user who hits your site with a 14.4 modem?

The following chapters introduce you to JavaScript syntax, procedures, and structure. Your left brain will eat it up. The purpose of this chapter and the Web surfing you're sure to do on your own is to build a well-balanced JavaScript brain: what the creative right-side can do with the left-side knowledge of JavaScript.

CHAPTER 2

Introduction to the Java Family

Welcome to the Java language family. The Java family is comprised of Java, Sun Microsystems' development environment and language, and JavaScript, an HTML (HyperText Markup Language) scripting tool developed by Netscape. These two languages support the development of dynamic content for the World Wide Web, and are built into the Netscape Navigator 2.0 browser (with full Java support expected in the 2.1 release for Macintosh). Because you are reading this book, you are obviously interested in what Java can do for you, and what it can do for your Web presence. In short—and forgive the melodramatics—the Java family brings your Web pages to life. Whether by means of a Java applet or a JavaScript function, many static, CGI script-based HTML pages are being replaced by pages that provide dynamic content and can automatically react to the user actions on the client side without reverting to server-side scripting.

The World Wide Web (WWW) is one of the fastest growing media for the transfer of multimedia information between computer users in homes, schools, businesses, and governments across the globe. What was even a year ago an arcane subject reserved for techno-gurus is now almost universally advertised. Many newspaper, magazine, and television ads now include WWW Uniform Resource Locators (URLs) for business home pages. With this increase in commercial usage comes the demand for more sophisticated presentations. The day of the text-based terminal is over. Full-motion video and interactive virtual reality with stereo sound are the holy grails sought by Internet-savvy companies these days. Much development work also is geared toward standardizing such media protocols as VRML and MPEG II. In the midst of the fray comes Java—a new programming language that promises to make much of the potėntial of the Internet a little closer to reality.

The following sections provide a brief synopsis and comparison of HTML, JavaScript, and Java, including what they bring to the Web and how they compare with each other. These languages comprise an entire package for developing dynamic, interactive Web content—from the page layout to the scripting of events and execution of full-blown applications. Although this book presents JavaScript to the new programmer who uses Macintosh computers, having this background will help you understand the larger importance of Java to multi-platform Internet content development, and make you sound more authoritative at water cooler bull sessions.

Netscape Navigator 2.0, with its integration of Java and JavaScript, provides a unified environment for dealing with the problems of creating dynamic client-side HTML pages. Therefore, in addition to discussing language specifications for writing programs in Java and JavaScript, this chapter discusses specific implementation details of these languages in Netscape Navigator 2.0 and how these features relate to running Java applets and JavaScript scripts.

The World Wide Web and HTML

The World Wide Web and HTML are a subset of the global network widely known as the Internet. The Web is popularly viewed as being synonymous with the Internet, because it is the most frequently used Internet protocol. In actuality, however, it is only one of many protocols used to deliver information over the Internet. A *protocol* in Internet terms is an agreed-upon format for transmitting information. Protocols specify the exact format of the data and how it is transmitted. Examples of such protocols are FTP, Gopher, HTTP (the World Wide Web), and email. To understand the role of JavaScript and Java as client-side execution languages, it is also important to understand the HTML protocol and the way it relates to the formatting and presentation of Web pages in a browser.

HTML is the first thing with which a Web developer should be thoroughly familiar when learning how to create a WWW presence. It is beyond the scope of this book to cover the entire HTML language in depth; therefore, if you don't already know how to create HTML pages, it is imperative that you pick up a comprehensive HTML book—such as *The World Wide Web Design Guide* by Hayden Books—and study it before attempting to take on JavaScript or Java. This book *will*, however, show you examples of how the HTML elements work whenever there is an instance in which JavaScript interacts with the HTML language. If you are familiar with HTML, you might already be acquainted with some of the following introduction to the Internet. Those of you who are unfamiliar, on the other hand, might be interested in the historical background of the Internet, World Wide Web, and HTML.

History of the Internet

In 1957, the U.S.S.R. launched the Sputnik satellite, much to the surprise of the U.S. government. Hoping to close the gap in the space race, the U.S. government created ARPA—the Advanced Research Projects Agency. The purpose of this agency was to research advanced technologies that were too risky for private enterprise to undertake, but were seen to be important steps in a technology-driven time. In 1969, the Department of Defense commissioned the ARPAnet to be created for research into networking protocols. Early on, it was realized that this network of computers would need a standardized protocol, so the TCP/IP protocol was created in 1973 to fulfill this need. This is the same protocol that still drives the transport of information over the Internet.

TCP/IP, or Transmission Control Protocol/Internet Protocol, is a collection of protocols that enable different networked platforms to interchange information. In today's computing environment, the hardware-based standard for routing network data is Ethernet; however, when TCP/IP was developed, there was no standard for networked computers. TCP/IP is considered a hardware-independent protocol that is carried along on top of whatever hardware-based protocol is being used. This means that you can use Ethernet, ATM, or whatever new hardware methods for transmitting data that are in use, or might be developed.

However, it wasn't the standardization of the protocol that popularized TCP/IP. In 1983, the University of California at Berkeley released a version of its UNIX operating system that incorporated the TCP/IP protocol. Because UNIX was running on many of the computers connected to ARPAnet, the TCP/IP protocol became the de facto standard for connecting to the ARPAnet. It is interesting to note that this scenario is very similar to the current relationship between Java and Netscape. Alone, Java might be a difficult sale, but by Netscape's inclusion of the Java run-time engine in one of the World Wide Web's most popular browsers, developers and supporters of the language have a guaranteed potential audience.

In 1983 there were 500 Internet hosts. In 1986, the National Science Foundation entered the fray by creating NSFnet, a backbone running at 56 Kbps that spanned five supercomputing centers around the country. By the end of the year, there were 5,000 Internet hosts. In 1989, with more than 100,000 hosts on the Internet, NSFnet was upgraded to a 1.5 Mbps T1 line. The next year, the role of the Advanced Research Projects Agency in the Internet was finished and ARPAnet ceased to exist, with all of its functionality being subsumed by the larger NSFnet and Internet in general. What was once a purely academic research project was now suddenly the next Big Thing on college campuses and large corporations.

NOTE Although it took 20 years for the original ARPAnet to reach the 1,000,000 hosts mark in 1992, within one year of reaching that mark the number of hosts doubled.

By 1992 there were more than 1,000,000 hosts on the Internet, and it was just beginning its meteoric rise. By the end of 1994, the Internet had more than 4,000,000 hosts connected, and in 1995, the NSF decommissioned the NSFnet. The role of the government in starting this fledgling technology was over, and private industry was expected to take up the slack. In March of 1995, the official number of com, or business addresses passed that of edu, or educational addresses for the first time, signaling the true beginning of a commercial model for the future of the Internet.

Development of the WWW

When the Internet began, it was widely perceived as an arcane and mysterious network useable only by UNIX gurus and the technically inclined. In many cases this was true. The protocols and commands were a dizzying array of cryptic codes for sending bits of information this way and that. However, there were attempts at making the Internet more useful, and applications such as email and newsgroups increased the everyday use of the Internet by non-technical users in the early eighties. Of course, these applications were all passive—the files being transmitted were static documents or images that were set once by the creator and typically did not change that often without significant work. The email file was just an electronic version of a letter, and newsgroups were simply electronic bulletin boards. As more and more users began to get onto the Internet, the demand for more and more useful programs grew.

In 1990, Tim Berners-Lee, from the European particle research laboratory CERN, developed a protocol for delivering different types of files over the Internet using a common protocol. This protocol was the beginning of the World Wide Web. At that time, the Web still had a text-based interface, and while popular, it didn't become the major force it is today until the National Center for Supercomputing Applications (NCSA) in Illinois developed a graphical interface for the Web. This interface was known as Mosaic.

Almost overnight the Web became the golden child of the Internet. It was impossible for Net surfers not to be swept away by its formatted text and fancy graphics—a far cry from what the Internet had been just a few years before. The text-based commands were gone; in fact, the command prompts themselves were gone! It was simply point, click, and poof, there it was—like magic. Of course, it wasn't exactly magic under the hood. Pretty soon, Web authors were incorporating more than just GIF images or audio files, and when they got away from the files that the browser could handle, browser users were forced to search the Net for a program that could display the latest cool media on their computer platform, often finding out that their operating system wasn't supported.

Despite these inconveniences, navigating the Web with Mosaic or one of the many other graphical browsers was worlds easier than it was before, and browser developers were constantly including more functionality for all platforms. Even though it was just point and click, the end user's mouse-clicking finger was still the most dynamic part of the system. The content provided was still as passive and static as ever—it was just getting easier to make it appear where you wanted it.

Features of the HTML Language

The new language that drove the World Wide Web's growth was the HyperText Markup Language (HTML). This language is a series of tags and directives inserted into a plain text file. These tags and directives indicate how the elements of the page are to be displayed. With HTML, you can do the following:

□ Create platform-independent documents

□ Create links to other documents on the Internet

□ Include graphics and multimedia

□ Link to additional resources on the Internet

The first and most important feature of the World Wide Web and HTML is that both are platform-independent. This means that the designer of the Web page can create one file that will work on all the PCs, Macs, and UNIX computers attached to the Internet. Because of the heterogeneous nature of the Internet, this feature was of primary importance. Because Web authors could not be sure what computer their file would be displayed on, depending upon platform-specific implementations would be too burdensome.

The ability to link to other documents is where the term *hyper* comes into play. *Hypertext* refers to the ability to create links in a document that send the reader to related information at the push of a button. This nonlinear approach to document design is intended to aid learning and increase interest in a subject by allowing readers to follow the paths they prefer, instead of relying upon the linear approach that has dominated text since the advent of writing.

In addition to displaying standard text in the browser window, HTML provides methods for including digitized media such as sounds, images, and movies. In addition, Netscape provides the capability to embed additional programs inside the browser window so that it can display more advanced media such as VRML and Shockwave Director files. By doing so, the HTML file is able to extend its functionality beyond the mere text representation used to create the HTML files.

The HTML language also can implement any new protocols that are added in the future. Because it uses a text tag format, any new tag can be ignored by a browser when it cannot handle it. This feature enables different Internet protocols to be called in the HTML file, thus adding to the extensibility of the language.

HTML has proved to be a highly extensible and versatile language for creating documents and presentations over the Internet. When HTML was first created, the Internet and the World Wide Web were still text-based, and many of the first Web browsers were text-only programs such as Lynx. In fact, in many pages you can see a "text only" link that enables users of these types of browsers to view HTML files that don't depend upon visual media such as image maps. It wasn't until Marc Andreessen, who later went on to start Netscape with Jim Clark, developed Mosaic at the NCSA that the Web was seen as a graphical medium. For a language to make the transition from simple text files to complex multimedia presentations, as HTML has done, is certainly a feat.

Today, there is less reliance on text as the principal content in HyperText Markup Language documents. Sure, the tags and directives are still there to tell the browser *how* to display the page, but they are more often displaying pictures, playing animation, or calling up an embedded, in-line program such as WebFX or Macromedia's Shockwave. However, even as complicated as modern Web pages can get, they are still created with the text-based HTML language that was used to create the relatively simple pages that got the WWW off the ground.

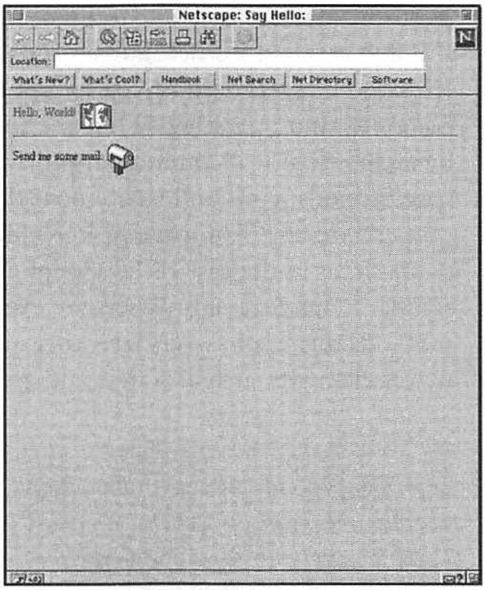
Hello, World! Page in HTML

Since the publication of Kernighan and Ritchie's *C Programming Language* book, a little program known as Hello, World! has been used to introduce individuals to new programming languages. In that tradition, as HTML, Java, and JavaScript are introduced, Hello, World! will be used in this book to show you how each language differs and how each compares. In the case of the version of the Hello, World! program in this text, a few extras will be added to further display the features of each language.

HTML works by telling the client browser, such as Netscape or the Microsoft Internet Explorer, how to display text, inline images, sounds, and other sundry file types that browsers and their helper applications now support. HTML performs this magic by using tags. Because you are most likely familiar with developing HTML pages, the details of how these tags work should not be necessary. Suffice it to say that these tags are interpreted by the browser and instruct it how to display the page for the end user. The following code is a simple HTML page that displays the words "Hello, World!"

```
<HTML>
<TITLE>Say Hello:</TITLE>
<BODY>
<A HREF="http://www.mcp.com/hayden">Hello, World!</A>
<IMG ALIGN=MIDDLE SRC="images/wrldbk.gif">
<HR>
Send me some mail: <A HREF="mailto: webster027@aol.com">
<IMG ALIGN=MIDDLE SRC="images/mail.gif" BORDER=0></A>
</BODY>
</HTML>
```

This file can be found on the CD-ROM in the Html subfolder of the JavaScript folder as Hello, World!.html. In Netscape 2.0, you should see what is shown in Figure 2.1.



Hello, World!

Send me some mail:

Figure 2.1 *The Hello, World! program as an HTML file.*

This short little program demonstrates each of the four main features of the HTML language that were previously discussed:

□ It is portable.

□ It uses media other than text.

□ It links to another site on the Internet.

□ It invokes a non-World Wide Web protocol.

The first thing this HTML file proves is that HTML is completely portable. This book will return to this issue again and again in dealing with technologies that will direct the future of the Internet. It is essential that these technologies be able to run on the diverse platforms that are available to users and that are connected and running on the Internet. Such is the case with HTML files. Files are saved in a standard ASCII format that is portable across all computer and operating systems. The tags themselves are defined by an international committee, and as long as you stick to their specifications, you can be confident that your document will display properly on any platform that implements the correct browser standards. If, as in the case of JavaScript and Java, you are not using the standard tags that all browsers are expected to handle, there are methods this book covers later that hide from unequipped browsers newer elements such as scripts and applets.

NOTE It is important to keep in mind individuals who don't have the means to view or execute the latest protocols and design your Web page with them in mind, whether you are inserting JPEG images, VRML worlds, or JavaScript programs.

Second, the Hello, World! HTML file brings in two images from the /images folder and tells the browser to display them in the document. In other words, the file deals with multiple media, and has the capability to direct browsers how to display them. It is important to notice that in the case of the HTML language, you need know nothing about the actual image itself. HTML defines only how to display the image, and depends upon the browser to understand the niceties of decoding GIF images and actually displaying them. In fact, the HTML language doesn't deal with the data at all—it simply indicates where the files can be found. You will see later on that the Java language overcomes this limitation and can actually perform the data manipulation and display that is required for more complex behavior such as image drawing. You will also see that JavaScript falls somewhere between the HTML and Java languages. JavaScript is actually more akin to the scripting qualities of HTML, although it retains the capability to call upon a more advanced programming syntax. At the same time, however, JavaScript is unable to implement Java's more advanced low-level behavior, such as direct display drawing.

The Hello, World! program also illustrates a third point about HTML. When you look at the Hello, World! page in your Web browser, you may notice that the Hello, World! words are highlighted. You should be familiar with the hypertext link that takes viewers to another location on the Web. This location could be a target in the same page, a new page on the current page's server, or a completely different site altogether. By using Uniform Resource Locators (URLs), HTML can dictate any file in the Web as a target as long as the file is accessible through the Internet. This is the hypertext portion of HTML.

JavaScript and Java both use tags similar to the HTML's HREF and IMG tags to indicate files to load into the browser. In the case of Java, these files are compiled programs—called *classes* in Java parlance—which load and then run on the client machine. In future releases JavaScript will have a similar option for loading scripts by using a URL. The ability to load scripts from the server and execute them on the client machine is a major element in the distribution of these new technologies. Instead of having the executable programs or scripts either running on another computer with the results being sent to you (as is the case with CGI scripts), or already existing on the client and running there, the Java and JavaScript files can be brought to the client over the Internet dynamically, and then executed locally.

Finally, in the case of the Hello, World! program, there are two images. The planet is a stand-alone image, while the mailbox is a link to send an email message. The HREF tag used to send the email message does not use the HTTP protocol, which is the transport method used by the World Wide Web. Instead, if you are using a mail-enabled browser such as Netscape, a mail composition window appears, and you can send an email to the address indicated.
This email message is then transported using the traditional email methods without using the HTTP protocol that runs the Web. This process keeps HTML from becoming overburdened with "feature-itis," as some refer to the ever-increasing size and functionality of modern applications. There was no reason for HTTP to implement its own mail system—it had one in place it could easily use. In many

ways this economy of features sums up the entire philosophy of the HTML language—*keep it simple and platform-independent.* The syntax itself is built around directing the display of many different media without regard for implementation-specific issues.

The Java Language

In April 1991, a small group of Sun employees moved off campus to Sand Hill Road, breaking direct LAN connection and most communication with the parent company. Settling on the name Green for their project, work began on what they considered a move into commercial electronics. In May 1995, Sun officially announced Java and HotJava at SunWorld '95. Over this four-year period, the Green group moved through consumer electronics, PDAs, set-top boxes, and CD-ROMs to emerge with a product that is the most likely contender as the ubiquitous programming language of the Internet in the next decade. What follows is a history of how the Java language came to be.

When the Green group was first envisioned as a foray into selling modern software technology to consumer electronics companies, it was realized early on that a platform-independent development environment would be needed. The public was not interested in what processor was inside their machines, as long as it worked well; in other words, developing for a single platform would be commercial suicide. James Gosling began Green's work by attempting to extend the C++ compiler, but soon realized that C++ would need too much work for it to succeed. Gosling then proceeded to develop a new language called Oak. The name came to Gosling when he saw a tree outside his window as he was entering the directory structure for the new language. After failing a trademark search, however, the name came to be known as Java.

Originally, four elements—Oak, an operating system known as the GreenOS, User Interface, and hardware—were put together into a PDA-like device known as *7 (star seven), named for the telephone sequence used to answer any ringing phone from any other in the Sand Hill offices. The small handheld device was good enough to impress Sun executives, but they were uncertain what the next step should be.

The technology in *7 was at first envisioned by the Green team as a marketable product that could be sold to consumer electronics manufacturers who would place the company logo on the front of boxes: a similar concept to what Dolby Labs had been doing for years. However, in early 1993 the Green team, now incorporated as FirstPerson, Inc., heard that Time-Warner was asking for proposals for set-top box operating systems and video-on-demand technology. These boxes would be used to decode the data stream that entertainment companies would be sending to consumers all over the country for display on television sets.

Ironically, at the same time FirstPerson heard about and began to focus on the set-top box market for interactive television, NCSA Mosaic 1.0, the first graphical Web browser, was released. Even as the Green technology was being developed for one market—set-top boxes—the field in which it would gain the most acceptance was just getting started itself. The Web had, of course, been around for several years by this time, developed at CERN by Tim Berners-Lee in 1990. Up to this point, however, the Web still retained the text-based presentation, which reminded people too much of UNIX and DOS. NCSA's Mosaic "prettied" the face of the Internet by enabling graphics and text to be merged into a seamless interface from what had been a cryptic and confusing system of protocols and commands.

Java and the Web were both developed at the beginning of the decade, an ocean apart. It would take another three years for the potential of the Web to be realized in Mosaic, and another two years before Java was made available to the wider Internet community.

At the time of Mosaic's release, FirstPerson was bidding on the Time-Warner TV trial, where hundreds of homes would be fitted with experimental video-on-demand hardware for testing. In June 1993, Time-Warner chose Silicon Graphics, Inc. over Sun. By early

1994, after a near deal with 3DO fell through, and no new partners or marketing strategy were forthcoming, FirstPerson's public launch was canceled. Half of the staff left for Sun Interactive to work on digital video servers and FirstPerson was dissolved. However, with the remaining staff, work continued at Sun on applying FirstPerson's technology to CD-ROM, online multimedia, and network-based computing.

At the same time that FirstPerson was losing the race for interactive television, the World Wide Web was winning the bandwidth race on the Internet. There was no doubt about it—the Web was big and getting bigger. In September 1994, after realizing the potential of Oak and the World Wide Web, Naughton and Payne finished WebRunner, later to be renamed HotJava. Soon, Arthur Van Hoff, who had joined the Sun team a year before, implemented the Java compiler in Java itself, where Gosling's original compiler had been implemented in C. This showed that Java was a full-featured language, and not merely an oversimplified toy.

Java took four years, and an evolution of purpose, to make it into the Internet mainstream. With Netscape Communications, maker of the popular Web browser Netscape Navigator, incorporating Java into its software, along with Java's potential in future applications such as intelligent agents and artificial intelligence, it is almost certain that Java is destined to be the most overarching technology of the Internet in the next decade.

Of course, Java's debut on the Internet is not the end of the Java mission. Sun sees Java's success on the Internet to be the first step in employing Java in interactive television set-top boxes, hand-held devices, and other consumer electronics products—exactly where Java began four years ago. Its portable nature and robust design enable it to be used for cross-platform development in competitive and unforgiving environments such as consumer electronics.

Features of the Java Language

The Java language (including JavaScript) changes the passive nature of the Internet and World Wide Web by allowing platform-independent code to be dynamically loaded and run on a heterogeneous network of machines, such as the Internet. Java provides this capability by incorporating the following features into its architecture. These features make Java a promising contender for being the major protocol for the Internet.

□ *Portable.* This means that it can run on any machine that has the Java interpreter ported to it. This is an important feature for a language to be used on the Internet, where almost any type of computer could be sitting at the receiving end of an Ethernet connection.

□ *Robust.* The features of the language and run-time environment make sure that the code is well behaved. This comes primarily as a result of the push for portability and the need for solid applications that won't bring down a system when a user accesses a home page with a small animation.

□ *Secure.* In addition to protecting the client against unintentional attacks, the Java environment must protect it against intentional ones as well. The Internet's law-abiding development community is all too familiar with Trojan horses, viruses, and worms created by a few outlaw programmers with malicious intent.

□ *Object-oriented.* The language is object-oriented at its foundation, and allows the inheritance and reuse of code both in a static and dynamic fashion. This news should excite the seasoned programmers among you.

□ *Dynamic.* The dynamic nature of Java, which is an extension of its object-oriented design, allows for run-time extensibility.

□ *High performance.* The Java language supports several high-performance features such as multithreading, just-in-time compiling, and native code usage.

- □ *Easy.* Because the language itself could be considered a derivative of C and C++, it is familiar to developers who currently use those languages. At the same time, the environment takes over many of the trickier tasks from the programmer, such as pointers and memory management. This point should further excite experienced, war-weary developers.

The job of providing dynamic content for the Internet is daunting, but the protocol that succeeds will become as universal as email or HTML is today. Java is the likeliest candidate so far to become just this protocol.

Java Is Portable

The Java programming language provides portability in several ways, including the following:

- □ The Java language is interpreted. This means that every computer it wants to run on must have a program to convert the Java codes into native code that particular machine understands.
- □ The Java language does not allow a particular machine to implement different sizes for fundamental types, such as integers or bytes.

By executing in an interpreted environment, the Java code does not have to conform to any single hardware platform. The Java compiler that creates the executable programs from source code compiles for a machine that doesn't exist—the Java Virtual Machine. The Java Virtual Machine is a specification for a hypothetical processor that can run Java code. The problem with traditional interpreters has always been their performance, or rather their lack of it. Java attempts to overcome this by compiling to an intermediate stage, converting the source code to bytecode, which can then be efficiently converted into native code for a particular processor.

In addition to specifying a virtual machine code specification to ensure portability, the Java language also makes sure that data takes up the same amount of space in all implementations. C programming language data types, on the other hand, change depending upon the underlying hardware and operating system. For example, an integer that occupied 2 bytes on a 68 KB Macintosh now takes up 4 bytes in PPC environments. The same problem exists across processor platforms, where some computers like the DEC Alpha are 64 bits, while others such as Intel's 486 are only 32 bits. By creating a single standard for data size, Java makes sure that its programs remain hardware-independent.

These are some of the features that make Java capable of running on any machine for which its interpreter is ported. This way, once a single application has been ported, the developer and user have the benefit of every program written for Java.

Java Is Robust

The Java environment is robust because it gets rid of the traditional problems programmers have with creating solid code. The Java inventors looked at extending C++ to include the functionality required by a distributed program, but soon realized that it would be too problematic. The two major problems in making C++ a portable program are its use of pointers to directly address memory locations and its lack of automatic memory management. These features enable the programmer to write code that is syntactically and semantically correct, and yet still proceeds to crash the system for one reason or another. Java, on the other hand, ensures a robust environment by eliminating pointers and providing automatic memory management.

Because the whole point of the Java programs is to be able to load and run automatically, it would be unacceptable for one of those applications to have a bug that could bring down the system by writing over the operating system's memory space, for example. For this reason, Java does not employ the use of pointers; a programmer cannot employ pointer arithmetic to move through memory. Additionally, Java provides for array bounds checking so that a program cannot access memory space not allocated to the array.

Java provides automatic memory management in the form of an automatic garbage collector. This garbage collector keeps track of all objects and references to those objects in a Java program. When an object has no more references, the garbage collector tags it for removal. The garbage collector runs as a low priority thread in the background and clears the object, returning its memory back to the pool either when the program is not using many processor cycles or when there is an immediate need for more memory. By running as a separate thread, the garbage collector can provide the ease of use and robustness of automatic memory management without the overhead of a full-time memory management scheme.

Java Is Secure

The necessities of distributed computing demand the highest levels of security for client operating systems. Java provides security through several features of the Java run-time environment.

□ A bytecode verifier

□ Run-time memory layout

□ File access restrictions

When Java code first enters the interpreter, before it gets a chance to run, it is checked for language compliance. Even though the compiler generates only correct code, the interpreter checks it again just to make sure, because the code could have been intentionally or unintentionally changed between compile time and runtime.

The Java interpreter then determines the memory layout for the classes.

NOTE The *class* is Java's basic execution unit—equivalent to an object in object-oriented programming parlance.

This means that hackers (the "outlaw programmers" I referred to previously) cannot infer anything about what the structure of a class might be on the hardware itself and then use that information to forge accesses. Additionally, the class loader places each class loaded from the network in its own memory area.

Even then, the Java interpreter's security checks continue by making sure that classes loaded do not access the file system except in the specific manner they are permitted to do so by the client browser or end user. Altogether, this makes Java one of the most secure applications for any system. Although site administrators may squirm in their Steelcase seats at the idea of programs fresh off an Internet site automatically loading and running on their network's computers, the Java team has made every effort to assure administrators that their worst fears, such as an especially effective virus or Trojan horse, will never become reality.

Java Is Object-Oriented

Java's most important feature is that it is a truly object-oriented language. The Java designers decided to break from any existing language and create one from scratch. Although Java has the look and feel of C++, it is in fact a wholly independent language, designed to be object-oriented from the start. This independence provides several benefits, including the following:

- □ Reusability of code
- □ Extensibility
- □ Dynamic applications

Java provides the fundamental element of object-oriented programming (OOP)—the object—in the class. The class is a collection of variables and methods, or functions, that is a self-contained blueprint for an object. This means that once the class has been created, it can be used as a template for creating additional classes that provide additional functionality. For example, a programmer might create a class for displaying rectangles on the screen, and then decide that it would be nice to have a filled rectangle. Instead of writing a whole new class, the programmer can simply direct Java to use the old class, with a few extra features (often referred to as a "subclass"). In fact, the programmer can do so without even having the original source code.

After a class has been created, the Java run-time environment allows for the dynamic loading of classes. This means that existing applications can add functionality by linking in new classes that encapsulate the methods needed. For example, you might be surfing the Net and find a file for which you don't have a helper application that will display it. Today, you would most likely use your favorite search engine to pinpoint an application that could display the file. A Java browser, on the other hand, can ask the server with the file for a class that can handle the file, dynamically load it in along with the file, and display the file without ever missing a step.

Java's High Performance

Typically the cost of such portability, security, and robustness is a loss of performance. It is unreasonable to believe that interpreted code can run at the same speed as native code; however, Java has a few tricks that reduce the amount of overhead significantly.

□ Built-in multithreading

□ Efficient bytecodes

□ Just-in-time compilation

□ The capability to link in native C methods

One way Java overcomes the performance problems of traditional interpreters is by including built-in multithreading capability. It is rare for a program to constantly be using up CPU cycles. Instead, programs must wait for user input, file, or network access. These actions leave the processor idle in single-threaded applications. Instead, Java uses this idle time to perform the necessary garbage cleanup and general system maintenance that causes interpreters to slow down many applications.

Additionally, the compiled Java bytecodes are very close to machine code, so interpreting them on any specific platform should be very efficient. In cases where the interpreter isn't going to be enough, the programmer has two options: compiling the code at runtime to native code, or linking in native C code. Compiling at runtime means that code is still portable, but there is an initial delay while the code compiles. Linking in native C code is the quicker of the two, but places an additional burden on the programmer and reduces portability.

Java's Ease of Use

Finally, the Java language is relatively easy for an object-oriented environment, while JavaScript is as easy as any full-featured, procedural programming language, such as BASIC. The Java language is simple and effective because of its well-thought-out design and implementation. The three most important elements that make it an easy language to use are as follows:

□ It should be familiar to many developers because it is fashioned after C++.

□ It eliminates problematic language elements.

□ It provides powerful class libraries.

Java is consciously fashioned after the C++ language, providing a look and feel that many experienced programmers are comfortable with. At the same time, Java eliminates difficult and problematic elements of C++ such as pointers and memory management. This means that programmers can spend less time debugging and more time developing functionality. Java also has a powerful set of class libraries that provide much of the basic functionality needed to develop an application quickly and effectively.

Hello, World! as a Java Applet

As you have read, the Java language is a far cry from the limitations of HTML. Java is a full-fledged language in its own right, and can provide capabilities unavailable to plain old HTML. The following source code implements the Hello, World! program again. This time, the additional images have been left out, and the program merely displays the text "Hello, World!" in the document. This simple program exhibits many of the features of the Java language.

```
import java.awt.Graphics

public class HelloWorld extends java.applet.Applet {
       public void init() {
              resize(150,50)
      }
      public void paint(Graphics g) {
              g.drawString("Hello, World!", 50, 25)
      }
}
```

Of course, this is not the only code we need. Notice the second line of the code:

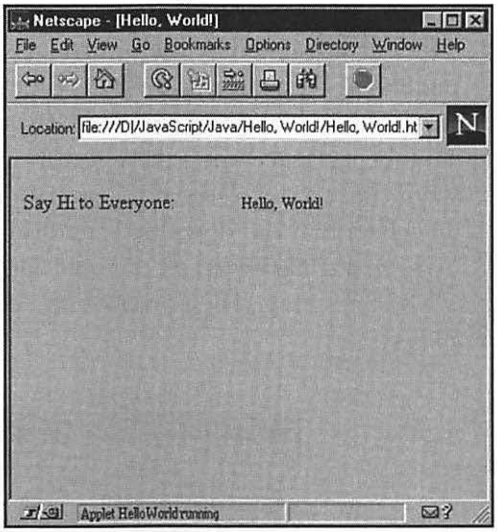
```
public class HelloWorld extends java.applet.Applet {
```

This code is saying to the compiler to create a class named HelloWorld that is a subclass of Applet. All applets expect to run in a particular environment, which is typically provided by a browser such as Netscape 2.0. This means that an applet needs to be called from an HTML file. The following listing is the HTML file that loads the Hello, World! applet.

```
<html>
<head>
<title>hello, world!</title>
</head>
<body>
say hi to everyone:
<applet align=middle code="helloworld.class" width=150 height=50>
➥</applet>
</body>
</html>
```

This file can be loaded into Netscape from a local disk by entering the path to the file with a leading file directive such as:

```
file:///CD-ROM/Java Examples/Hello, World!/Hello, World!.html
```


Netscape - [Hello, World!]
File Edit View Go Bookmarks Options Directory Window Help
N

Location: file:///D|/JavaScript/Java/Hello, World!/Hello, World!.ht

Say Hi to Everyone: Hello, World!

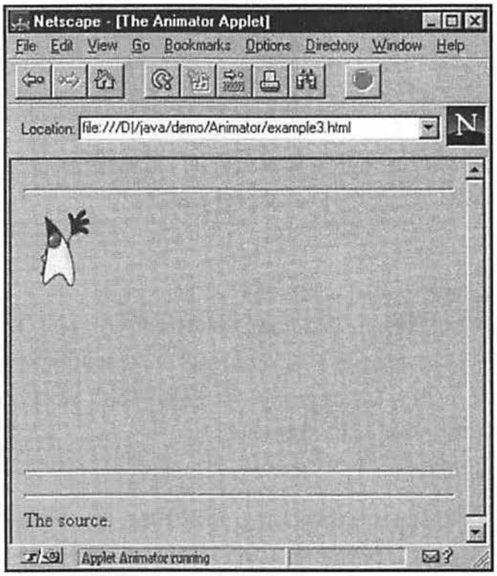
Applet HelloWorld running

Figure 2.2 *The Hello, World! program as a Java applet and HTML file.*

NOTE Please note that the screens shown in Figure 2.2 and in Figure 2.3 are from the Windows 95 version of Netscape 2.0. At the time of this writing, Netscape 2.1 with Java applet support for Macintosh was not yet available.

Pretty captivating, eh? Okay, so maybe it is a little less snazzy than the previous HTML-only page, but because the purpose of this example is to introduce each language, it wasn't meant to be snazzy.

The fact that this particular Java applet does not draw pictures gives a good indication of one feature of Java: as a language, it is very low level. Whereas with HTML, a simple tag pointing to the image file is enough to get it to display; with Java, because it is working at such a lower level, to program that behavior from scratch, it takes a much larger example than belongs in an introductory chapter. At the same time, though, the features of the Java language enable you to build upon the work of others easily, while not limiting yourself to a particular browser implementation. When the HTML Hello, World! picture was loaded using the HTML file, all of the functionality was subsumed by the browser. As Web page developers, we depend upon the features we expect everyone's browser to have, such as the capability to display GIF and JPEG image formats. With Java, dependence upon these features is not necessary. A Java program that creates a repeating animation will work on any system, anywhere that can download and display Java applets in HTML documents. However, an example applet by Sun that does just this is 883 lines long. The power of Java comes at the expense of complexity.


Netscape - [The Animator Applet]
File Edit View Go Bookmarks Options Directory Window Help
N

Location: file:///D|/java/demo/Animator/example3.html

The source.

Applet Animator running

Figure 2.3 *The Animator Applet from the Sun Java Development Kit in Netscape 2.0 for Windows 95.*

Even though Java is complex, at the same time it is easy. Easy, that is, for the author of an HTML page that uses a finished Java applet. Take a look at the HTML file example1.html in the java\demo\Animator directory of the CD-ROM:

```
<title>The Animator Applet</title>
<hr>
```

```
<applet code=Animator.class width=100 height=200>
<param name=imagesource value="images/Duke">
<param name=endimage value=10>
<param name=soundsource value="audio">
<param name=soundtrack value=spacemusic.au>
<param name=sounds
➥value="1.au¦2.au¦3.au¦4.au¦5.au¦6.au¦7.au¦8.au¦9.au¦0.au">
<param name=pause value=200>
</applet>

<hr>
<hr>
<a href="Animator.java">The source.</a>
```

As you can see, after the class file is created, it can be used in eight lines of HTML code, as compared to the 883 lines needed for the class file itself. The page author simply provides the images and sounds, and tells the applet what to do. For the HTML-only author, Java class files are as easy to use as any other media.

This book's coverage of Java will focus solely on its role in the area of applet creation and use, and how this role can be tied in with the JavaScript language. Java has a wider functionality associated with stand-alone programs, but for the Web page designer, this feature will be less important than the applet functionality that is implemented by the forthcoming Netscape 2.1 browser for the Macintosh. It is for this reason, therefore, that the stand-alone functions of Java will not be considered.

The biggest feature of the Java language to keep in mind is its object-oriented nature. In the program listing for Hello, World!, you should have noticed some organizational features of the program. In essence, you have a class, which is Java's representation of an object declared by this statement:

```
public class HelloWorld extends java.applet.Applet {
...
}
```

The declaration `public class HelloWorld` tells the Java compiler to create a new class or object that is based upon the applet object. Suffice it to say that Java is an object-oriented programming language at its core, and all Java programs are required to follow this methodology. Thus, our Hello, World! program must define itself as an object. Everything inside the outside braces are what defines our Hello, World! applet, and this is the basic building block for any Java class.

Inside the object definition, you can see two additional blocks defined by the curly braces:

```
public void init() {
      resize(150,50)
}

public void paint(Graphics g) {
      g.drawString("Hello, World!", 50, 25)
}
```

These two statements are known as ***methods*** in Java. In addition to methods, Java objects or classes can also have variables or containers for information. When we look at JavaScript, we will see that JavaScript objects have the same properties. However, where in Java objects are a requirement, JavaScript is ***much*** more lenient, and will allow more standard procedural styles of programming. BASIC programmers and others who don't currently "cut code," breathe a sigh of relief. And take heart; the following section covers this book's hero: JavaScript.

The JavaScript Language

The new arrival in the Java family is JavaScript. This scripting language has been implemented for the first time in the 2.0 version of Netscape, and aims to provide a compact, object-based language that is used to both create dynamic stand-alone scripts that can be embedded in HTML pages and interact with Java applets, linking together objects in a page. In addition, JavaScript will be available for server-side scripts in much the same way as Common Gateway Interface (CGI) programs are used today. This functionality will provide HTML authors with the flexibility to create powerful presentations without relying on a complicated system of CGI programs and server calls to process user input.

JavaScript, while linked with Sun's Java programming language, was actually created at Netscape, and is in reality a completely separate language from Java. In its first beta release, the language was referred to as LiveScript—relating the language to the LiveWire server protocol that Netscape was developing for its line of HTTP products. In fact, you may still see it referred to as LiveScript in some documentation on Netscape's site. Soon, it became apparent that Java was poised to take over much of the job being performed by server-side scripts and programs, and that a client-side scripting tool would be useful. Because LiveScript fit that bill nicely, Netscape partnered with Sun and renamed the language JavaScript, and began to market it as the companion scripting tool for the Java language. The main connections between Java and JavaScript are the similar syntax and the capability of JavaScript to access the public members of a Java class. JavaScript also is a very useful language on its own, and can be used to create complex programs that can interact with the user without relying on network transmission to run programs using the CGI method.

The JavaScript language is an interpreted script that is part of the standard HTML file. When the file is loaded, the Netscape 2.0 browser interprets the script and performs the operations specified. JavaScript works only on browsers that implement the JavaScript engine, and right now Netscape 2.0 is the only program that does this. Netscape has been a convincing leader in the browser market, and the role that JavaScript fills in relation to the Java language is an important and potentially lucrative one. Therefore, you can expect other companies to bring JavaScript into their browser implementations as well. Indeed, Microsoft announced in December of 1995 its support for Java in its Internet Explorer browser.

Features of the JavaScript Language

The development of JavaScript necessitated a balance between a full-blown language, such as Java, and a simple scripting language for controlling browser functionality, such as HTML. Of course, the JavaScript developers did not want to compete with either of these standards, and instead made their language do a good job of carrying out the tasks that fall between these two implementations.

The JavaScript language has several important features that make it a powerful addition to client-side Web development.

- □ *Simple.* JavaScript is simple and based upon the syntax of Java, which makes it an easy language to learn for those who are used to Java or C++ already, and not too hairy for HTML-only folks. It is also a good stepping stone for those wishing to learn Java after JavaScript, as presented in this book.
- □ *Dynamic.* JavaScript is dynamic and can react to user and client system input without recourse to server-side programs, such as CGI scripts.
- □ *Object oriented.* JavaScript is object-based, which means it can implement its own objects. It can also interact with objects in the browser and other object plug-ins. In future releases, JavaScript will be able to interact with Java applets that are loaded on a page-by-page basis.

The combination of features upon which JavaScript is built define it in relation to HTML and Java, each of which has either already set itself as the de facto standard for Web page creation (HTML) or has made impressive strides toward capturing the high ground of Internet application delivery (Java). It is important to understand these features and be able to weigh them against your own Web requirements when designing, building, and maintaining a Web presence.

JavaScript Is Simple

The JavaScript language can be seen as a simple and compact design that takes on the basic syntax and control-flow structure of Java to facilitate a quick transition between the languages. For example, all of the control-flow statements that JavaScript uses are present in Java such as `if…else` and `while` and `for` loops. In addition, all of the expressions for arithmetic, logical, and string statements are present in JavaScript from Java. This means that programmers are able to use a common language set for creating both Java applets and JavaScript scripts.

JavaScript is not a strictly typed language, and therefore variables are not declared as in Java. In Java, whenever you create a variable to hold a value, you must tell the compiler what type of variable it is. From then on, the compiler will check to make sure that the variable stays as that type. This is known as *strict type checking*. For example, the following code creates two variables in Java. The third line generates an error because the two types of variables are not the same.

```
int number = 10   // create a variable which holds a number such as
➥1, 5, 100, 32,056 …
String text = "20" // create a variable which holds a String such as
➥"Hello, World!"
number = letter       // this line would produce an error
```

This type of strict type checking is not present in JavaScript; a variable can be freely associated with any type of data. When declaring variables, there is no need to indicate what type of variable it is to be. This freedom has both benefits and drawbacks, but for most scripting, the strict type checking of Java would be unnecessary.

```
number = 10   // create a variable and assign it the number 10
text = "20"   // create a variable and assign it the string "this is
➥a test"
number = text             // this would convert
```

In addition to loose type checking, JavaScript also does not enforce the creation of classes and an object-oriented paradigm. This enables the programmer to use the language in the most efficient way possible without resorting to more complicated class hierarchies and inheritance. Instead of creating a class, member methods, and variables for a very simple function, the programmer can simply write the function and call it when its capabilities are needed. This behavior will be detailed in Chapter 4, "Control Flow and Functions in JavaScript."

JavaScript Is Dynamic

One great benefit of JavaScript is its capability to elegantly handle dynamic events in its environment. By interacting with the graphical form input built into HTML and the Netscape objects available to JavaScript, the programs created can react to user input in an event-driven manner, much like a stand-alone Macintosh application. This capability is an important benefit in developing client-side behavior without too much complexity or reliance on server-side network calls.

JavaScript Is Object-Based

While not a truly object-oriented programming language in the strictest sense, JavaScript can be considered object-based—that is, much of its functionality comes from the interaction of the scripts with objects that have exposed their methods to the scripting environment. These objects include the window, location, history, and document objects that Netscape provides, any applet objects that are loaded in a document, and any in-line programs that have been integrated with the Netscape browser. By interacting with these objects, the JavaScript language is able to incorporate the powerful capabilities these advanced programs can provide.

NOTE The JavaScript version implemented in Netscape Navigator 2.0 does not enable interaction with Java applets. However, future releases are expected to have this functionality.

The objects provided by Netscape to the JavaScript interpreter enable programmers to gather information about the execution environment and alter the way in which documents are displayed accordingly. For example, you can access the document history of the browser session and control where to go next depending upon where the user had been. Additionally, you can control the display characteristics such as the background color. Whereas with HTML you are stuck with one setting per file, with JavaScript you can make these changes immediately without requesting a new page from the server.

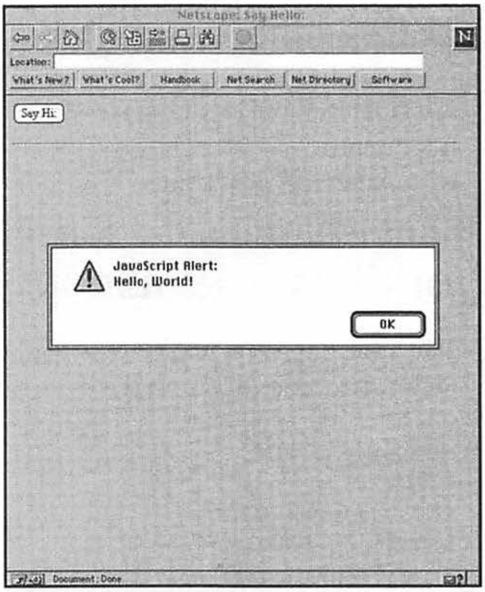
In addition to built-in objects, you will also be able to access applets and in-line programs that might be loaded by an HTML document. For example, you could load an animator applet, and call its `stop()` or `start()` method, depending upon events occurring in a form. You could alter the animation an applet was playing depending upon which form item had the focus, displaying pertinent information for that control. This enables you to use the simple form creation methods available in HTML and link them to a more complicated behavior provided by the Java applet. You wouldn't necessarily have to know how the Java applet worked beyond the functions you called to get the behavior you desired. This capability to script together objects in an easy fashion is one of the more powerful features of the JavaScript language. As mentioned before, this is a feature slated for incorporation in future releases. For now, you can still load applets to be run when a page is opened, but you cannot interact with them in any way beyond what is already available with standard HTML tags.

Hello, World! as a JavaScript Function

The following code implements the Hello, World! program in JavaScript style, and shows off the dynamic ability of the language.

```
<HTML>
<HEAD>
<TITLE>Say Hello:</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--  this line hides the script from old browsers
       function helloWorld() {
              alert("Hello, World!")
      }
// this is the end of the script and comment structure-->
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="button" VALUE="Say Hi:" ONCLICK="helloWorld()">
</FORM>
<HR>
</BODY>
</HTML>
```

As you can see, the code is integrated into the HTML file. In a way, JavaScript can be considered a type of HTML, but the behavior it invokes is much more complicated. Notice the creation of a function known as `helloWorld`, which makes a call to a function named `alert`. The `helloWorld` function can then be used anywhere in the HTML file to enact the behavior specified in the function declaration. In this case, the button created by the HTML code calls the function `helloWorld` when it is clicked. The `helloWorld` function then calls the method `alert()`, which brings up a dialog box with the string indicated in the method call. Figure 2.4 shows the resulting display.



JavaScript Alert:
Hello, World!

Figure 2.4 *The Hello, World! program as a JavaScript function.*

In the next chapter, "JavaScript Fundamentals," you will learn right away the specific details of how a JavaScript program is put together and how to begin using JavaScript code in your own HTML pages.

A Comparison of Java and JavaScript

Now that you have gone over each of the languages in general terms, there are more specific implementation details that can be seen in comparing Java and JavaScript. Table 2.1 shows the different ways in which Java and JavaScript have been implemented.

Table 2.1 *A Comparison of Java and JavaScript*

Feature	Implementation
Execution	**Java:** Java source code is compiled before it is sent to the client to be executed. There still has to be an emulator or interpreter to execute the file on a specific platform.
	JavaScript: JavaScript is not compiled at all before it is sent to the client; rather, it arrives as a plain text file, to be interpreted line by line in the browser.
Object methodology	**Java:** Java is a truly *object-oriented* programming language, and the programmer is required to develop objects even for the simplest programs.
	JavaScript: JavaScript is an *object-based* language that does not have classes or inheritance. However, objects of a sort can be created and the language can interact with objects created in other programming languages.
Use in HTML	**Java:** Java is a separate format from HTML and must be loaded like an in-line image or sound file.
	JavaScript: JavaScript is plain text and can be either integrated into the HTML file directly or embedded and loaded dynamically. Either way, the resulting code is still in a text format.
Variable declaration	**Java:** Java uses strong type checking, and all variables must be declared at compile time before they are sent to the client machine.

continues

Table 2.1 *Continued*

Feature	Implementation
	JavaScript: JavaScript uses loose type checking in that the interpreter evaluates the data type at runtime and makes the appropriate assignments.
Object references	**Java:** Java object references must be available at compile time for the compiler to be able to implement strong typing. This is referred to as static binding.
	JavaScript: JavaScript object references are left to be checked until runtime. This is because without compilation there is no way to even implement object reference checking. This is considered dynamic binding.

In addition to the previous features, it is worth noting that both Java and JavaScript implement a strong security philosophy, and neither are allowed to access the end user's local hard disk.

Netscape 2.0 and the Java Family

Java and JavaScript are currently implemented only in Netscape Navigator 2.0 for Windows, but many companies have indicated support for these standards. Java is in a more mature state than JavaScript, and has therefore more support in the development community. Companies such as Metrowerks, Natural Intelligence, Tradewave, Microsoft, Symantec, and others are all busy developing Java-enabled browsers and development tools. By the time you read this, there may be other companies which have released versions of browsers that are Java-enabled. Of course, Sun is working on its own browser, HotJava, which will support Java and JavaScript.

Because JavaScript is in a much earlier stage, there are no firm commitments yet by companies promising to implement JavaScript in their browsers. As the role of Java burgeons on the Internet, however, so too will the need for such a scripting language, and it can be expected that JavaScript will take on a similar importance. For now, Netscape 2.0 is the only game in town that plays by JavaScript rules on the Macintosh.

Built-In JavaScript Interpretation

The JavaScript engine is built into Netscape 2.0 for every platform, from PC to Mac to UNIX. If you are running Netscape 2.0, you can run JavaScript programs. Netscape's JavaScript implementation is of course the de facto standard now for the language. The JavaScript specification is being placed before Internet committees in an attempt to position it as an open standard on the Internet. This means that other companies will be likely to implement it because they are freed from licensing fees to Netscape.

Built-In Java Run-Time Engine

The Java run-time engine-enabled Netscape 2.0 is currently only available in 32-bit Windows (that is Windows 95, Windows NT) and UNIX versions. It does not support 16-bit Windows (that is Windows 3.*x*) nor the MacOS. Although a 16-bit version for Windows is not expected, work is underway at several companies, including Sun, to develop a Macintosh version. Sun released a beta version of the Java Development Kit for Macintosh in February. You can download this kit in its latest version from http://www.javasoft.com.

Summary

The functionality and versatility of Java and JavaScript for the Web site developer are bound to make them a cornerstone of the next generation of Internet content development. Each language brings unique features to the Web and to the development of dynamic, distributed applications that can make the Web come alive.

Java is a powerful and feature-rich specification that attempts to overcome the many hurdles in the way of distributed, object-oriented computing. Java has attained the portability, security, and robustness necessary, while at the same time keeping the language easy to use and at a high performance level. Java, from its beginning more than four years ago in consumer electronics, has become a powerful tool that encompasses many of the advancements of computing in recent years. Object-oriented, dynamically extensible, multithreading, and robust—Java fits in well with the emerging realities of the Internet and personal computing systems.

JavaScript supports the object-oriented features of Java and other Netscape plug-ins and provides the means for you to stitch together diverse components to create a rich online presence. In addition, JavaScript is capable of quite powerful behavior and can bring HTML documents to life by incorporating event-driven scripts. The ease of use of the language also means that anyone who can already write HTML can migrate to JavaScript easily, and then move on to Java itself.

It is the aim of this book to enable the competent HTML author to learn the complete capacity of JavaScript and to briefly introduce the next logical steps that lead to Java development from JavaScript. In the next chapter, the fundamentals of the JavaScript language are presented. This introduction takes you through the syntax of the language and prepares you to place JavaScript into your own HTML documents. In later chapters, you will use this information to bring in more JavaScript functionality. Many examples are provided along the way, but, in the spirit of Web development everywhere, you need to get out on the Web and read other JavaScript developers' source code! You can learn a lot by seeing how other, everyday citizens of Gotham are wiring up their own sites with cool scripts.

CHAPTER 3

JavaScript Fundamentals

Chapter 2 of this book introduces the reader to both the Java and JavaScript languages, their development, purpose, and design. It is important to understand these issues when picking up a new programming language, because using new programming language families effectively begins with knowing when to speak which dialect. Of course, JavaScript is a derivative (or a simpler dialect) of Java, so many of the lessons you learn here are applicable if you decide to move on to Java in your future work. Also, a little history never hurts, does it? From here on out, though, we will zero in on JavaScript and the elements that make up the scripting language as found in Netscape. Now would be a good time to power up Netscape 2.0 on your Macintosh, especially if you're the curious type who cannot wait to try some test scripts and see what Netscape does with them.

This chapter provides fundamental programming lessons for learning JavaScript. To program in JavaScript, you must know how to get JavaScript programs to run by doing the following:

- □ Learning how to integrate JavaScript into your HTML code
- □ Learning the syntax and general structure of the JavaScript language

After you become familiar with what a JavaScript program looks like and have learned how to properly place one into an HTML document, you are ready for this chapter, which covers the following elements:

□ Creating variables

□ Using variables in different expressions

□ Using operators with variables

Knowing how to create variables, form expressions, and use operators, as well as understanding the basic elements of style, provides you with all of the building blocks you need to begin coding useful JavaScript statements that can be incorporated into more complicated scripts discussed in later chapters. (Isn't it great how books like this one *always* save the really juicy stuff for "later chapters"?)

NOTE In many of this book's sections, you will run across places where elements of style in programming are discussed. Elements of style are not imperative to programming a JavaScript or Java program, but they become increasingly important as you look toward code reusability and maintenance, or not having fellow Web programmers scoff at your scorn for readability. Although high-level languages such as Java and JavaScript are designed to be easily read and understood by programmers (at least in comparison to older low-level languages, such as assembler), creating easy-to-read code is more the result of a programmer's good habits than of any language's features. Hopefully you will find the style advice useful and portable to any language to which you may move in the future. Don't forget: On the Web, *anyone* can read your HTML and JavaScript code by simply choosing View Source or a similar command in his or her browser. So be on your best behavior!

JavaScript and HTML

JavaScript is interpreted from HTML files that are loaded into the Netscape 2.0 browser. This means that to use JavaScript in your Web documents, you must be able to integrate your JavaScript code with the rest of the HTML code. You can include JavaScript code in a document the following two ways:

- □ Embed the code within the document
- □ Load the code from a separate file

In addition, you need to learn how to use HTML to hide JavaScript code from older Web browsers that cannot properly process it. You also need to understand the order in which JavaScript code is executed.

Embedding a Script within an HTML Document

To embed a JavaScript program into your HTML file, use the <SCRIPT></SCRIPT> tags and place all your code between the beginning (<SCRIPT>) and end (</SCRIPT>) tags. Although the <SCRIPT></SCRIPT> tags will gain several options in future JavaScript releases, for now you only need to specify what scripting language is being used. In this case it is, of course, JavaScript:

```
<SCRIPT LANGUAGE="JavaScript">
</SCRIPT>
```

After you have created the tags, any code that is inside is interpreted by the JavaScript interpreter. For example, the following HTML file uses JavaScript for all of the text output to the screen.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
document.write("<HR>")
document.write("This is a line written to the browser window by
➥JavaScript.")
document.write("<HR>")
// -->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

In this example, the JavaScript source is placed in the <HEAD> </HEAD> tags of the HTML document. This is not necessary; placing the <SCRIPT></SCRIPT> tags in the body would work just as well. However, unless you are trying to mix JavaScript output with HTML that's between the <BODY></BODY> tags, your code will be more robust if you define the functions in the head of the HTML document. When you begin creating JavaScript functions and calling them with event handlers, the functions will be read in before any HTML code that might attempt to access them. If you are using the JavaScript code to format the page, you might need to place it in the body; otherwise, try to keep JavaScript elements in the head portion of an HTML file.

Loading Script Source Files

NOTE **Although this feature is documented in the JavaScript guides, it is not currently implemented in the Netscape 2.0 browser for Macintosh or Windows 95. It's worth keeping in mind, however, for future releases.**

Another useful way to load JavaScript source code into a page is to load a text source file that holds the JavaScript source code. To load the source file, instead of writing JavaScript source code directly into the HTML file, you can provide the URL to the file that holds the source code:

```
<SCRIPT LANAGUAGE="JavaScript" SRC="JavaScriptCode.js"></SCRIPT>
```

You should remember the following two things when importing JavaScript code:

□ The extension to the source code file should be .js

□ The LANGUAGE attribute is not necessary if you indicate the source type in the extension

Therefore, the following code is equivalent to the previous statement:

```
<SCRIPT SRC="JavaScriptCode.js"></SCRIPT>
```

The advantage to using source files instead of embedding the code inside your HTML files is that you can update the source code without changing the HTML file at all. You simply replace the old .js file with the replacement copy. Of course, you must make sure that the new copy works with the original HTML file.

Commenting out JavaScript Source Code

You might have noticed two codes that were inside the <SCRIPT></SCRIPT> tags, but outside the actual scripting code:

```
<!--
// -->
```

These two lines are HTML comments that hide the JavaScript source code from browsers that are not able to interpret the <SCRIPT></SCRIPT> tags. Everything between the comment tags will be ignored by older browsers. You should note the two forward slashes (//) before the end comment tag (-->). These are actually JavaScript comments that hide the end HTML comment tag from the JavaScript interpreter. It is useful to mark these lines with explanatory comments.

```
<SCRIPT LANGUAGE="JavaScript">
<!-- this line hides the JavaScript source from older browsers
document.write("Hello, World! - try this on older browsers with and
➥without HTML comments")
// this is the end of the HTML comment tag -->
</SCRIPT>
```

Additional HTML Considerations

There are additional considerations when dealing with the Netscape Navigator 2.0 edition of JavaScript and HTML files. While these are not features of the language, they arise from the interaction of JavaScript in the Navigator environment and mean that typical HTML code might require some special attention.

First, <IMG> tags in HTML files can cause problems if they do not have their WIDTH and HEIGHT attributes set. For this reason, it is always preferable to set the width and height values for images you use in your program. If you include a <SCRIPT> tag after the last image tag, this also fixes the problem.

Second, nested tables and forms can cause problems with JavaScript, so if you use tables to organize your forms, you might want to rethink the layout of your tables if you have created a nested hierarchy. In Chapter 8, "The Forms, Window, History, and Location Objects," forms are covered in more detail, but because you most likely already have HTML files you want to incorporate JavaScript into, you might need to think of how the HTML will fit in with the JavaScript structure.

Third, there are several functions in JavaScript that enable you to send text output to the screen, including returns, tabs, and spaces. Traditionally, HTML browsers ignore any formatting information of this kind unless they are surrounded by the <PRE></PRE> or <XMP></XMP> tags. Be sure to use these wherever you have JavaScript output.

Processing Order

JavaScript code is interpreted within the HTML page after the entire page loads, but before the page is actually displayed. When you use source files, and call them in as SRC attributes in the <SCRIPT></SCRIPT> tags, this code is evaluated as if it were script container content. This evaluation occurs, however, before any embedded scripts are executed.

Later on, when the use of functions is discussed, this book presents a more in-depth look at when scripts are evaluated. For example, functions within scripts are stored but *not executed* until they are called by an event handler. This can create interesting behavior that can either work for or against you. Be sure to check out the section on processing order and event handlers in Chapter 4, "Control Flow and Functions in JavaScript."

JavaScript Architecture and Writing Code

The JavaScript architecture is based on tokens that the interpreter extracts from the source file and executes. A *token*, as used in most programming languages, is any indivisible unit that occurs in a program and is distinguishable from other objects. Examples include reserved words (`if`, `for`, `next`, `while`), operators (+, -, /, *), and identifiers (the name "counter" assigned to an integer variable). The following list contains JavaScript's basic tokens:

□ Comments

□ Literals

- □ Identifiers
- □ Separators
- □ Operators

These elements of the JavaScript language provide the basic building blocks of all code you will ever write. You can imagine all of these tokens as different bricks, beams, shingles, nails, glues, and board that can be combined into an almost infinite combination of buildings. Knowledge of how each of these elements works with the others enables you to write successful programs. How solid, yet elegant, your building—your program—is depends upon your skill as an architect.

NOTE This chapter contains the type of syntax you will use when typing JavaScript code for things such as numbers, variables, function names, comments, and so forth. Many errors you will run into later when creating your own programs will come from typos and forgetfulness. The problem is that most of the information provided is relatively simple, and you might be tempted to gloss over it. You should still take time to read and understand everything. The programming exercises in later chapters of this book will be much more enjoyable if you have retained the material presented in the earlier chapters, and you won't have to keep flipping back and forth in the book to remind yourself of something. Practice, practice, young Grasshopper.

Comments

When writing code in any language, it is important to remember to include written comments about what each line or group of statements is doing. To keep the JavaScript interpreter from trying to execute this text, you must hide it between comment characters. Comments in JavaScript come in two styles. Table 3.1 presents the two types of comments available in JavaScript.

Table 3.1 *The Different Comments in Java and JavaScript*

Comment Type	Description
// comment	All characters after the // to the end of the line are ignored.
/ comment */*	All characters between /* and */ are ignored. These comments can extend onto multiple lines.

The older (/* */) style was originally used in the ANSI-C standard and is carried on (rather nostalgically) in Java and JavaScript. The text that comes between the front comment marker (/*) and back comment marker (*/) is hidden from the interpreter during execution. The /* */ comment is useful for multiple-line comments found, for example, at the beginning of a code fragment, such as the one that follows:

```
/* The following JavaScript function converts
   from Fahrenheit to Celsius.
   created 11-Jan-95 by MGJ                    */
function fahrToCel(fahrTemp) {
    celsTemp = (fahrTemp-32)*5/9
    return celsTemp
}
```

If you need to comment out only a single line, there is a more useful syntax. The // comment method was added to C++ as an easier way than the older C style comment (/* */) to comment out individual lines, and it is retained in JavaScript and Java. You can either use this to comment out an entire line...

```
// this line returns the Celsius temperature
return celsTemp
```

... or just the last portion of a line:

```
return celsTemp  // return the Celsius temperature
```

NOTE If a line of code does something important, it is useful to call attention to it with a short comment. It is always considered "good programming" to provide comments for all your code to help future programmers (including yourself) understand what's happening.

The type of comment you use and for what purpose you use it is entirely up to you, the programmer.

Two styles of commenting programs are the Sun and Microsoft styles. Most of the Sun documentation uses the following style for multiple lines, because it is useful with the automatic document generator that comes with the Java Development Kit:

```
/**
 * this is a multi-lined comment
 * for the following function
 */
```

The `/**` tag on the first line tells the document generator to use the comments as text in the resulting HTML file that it creates automatically from source code. This type of comment is not used in JavaScript, but if you are moving on to Java, it is nice to get into the habit of using it. It doesn't affect the comment, but when you begin coding Java applets and want to comment a line, the automatic document generator will use your comments to generate the text of the resulting HTML file. As with the standard comment, `*/` closes the comment block.

Microsoft program documentation uses the `//` comment in code, especially the code that is associated with the Visual C++ environment. The following code is an example of a multi-lined comment in Microsoft documentation.

```
///////////////////////////////////
// this is a multi-lined comment
// for the following function
```

Again, the choice to use either style is up to you. If you are planning to use Java extensively in the future and are using JavaScript as a stepping-stone to learning Java, you should probably use the Sun form for comments, because it will be helpful when you later make the transition to Java. If you are intending to use only JavaScript, the // comment is easier to use because it simply comments out the rest of the line, and you don't have to keep track of the beginning (/*) and end (*/) tags.

Including comments in code is an important habit to develop. Even though it seems as though everything is perfectly clear when you are writing your code, someone else who wants to use the code might have great difficulties in determining exactly what certain lines of your code are supposed to accomplish. Using comments also saves you many headaches later when you decide to update a JavaScript function that you spent weeks creating, only to find that you can't figure out how to work in new code without disrupting what you've done.

Literals

Literals, which are used when entering explicit values into your code, such as the number 5 or the word "Chicago," refer to the basic representation of these two types of data in the JavaScript language:

□ Numbers

□ Strings

Number literals can be categorized into two subtypes: ***integers*** (numbers without a decimal point) and ***floating points*** (numbers with decimal values). In addition, there is a special literal known as the boolean. A ***boolean variable*** can be either true or false. The ***boolean literal*** is included under numbers because in C there was no true boolean literal; instead, the integers 1 and 0 were used for true and false respectively. Therefore, although boolean is being categorized as a number here, it really should be considered its own fundamental type. The string literal is any character sequence that is placed between single (') or double (") quotes. So, segments like "hello," "testing, one, two, three," and 'a' are all string literals.

NOTE Literals are closely linked with types, which are discussed in the section on declaring variables. The difference between types and literals is that literals are explicit values entered into the code of your program, whereas types refer to the type of internal representation and storage that the JavaScript interpreter assigns to certain types of data.

Think of the child's toy that has different pegs to put into the properly shaped holes. It has round, square, triangular, and star-shaped pegs, and a set of holes to match. The holes that can accept the different pegs are the equivalent of types—they are the variables that hold the data. Literals are the equivalent of the pegs, or data, actually placed in those holes, or variables.

Integer Literals

The first literal is the integer. This literal, the most common of literals, comes in three guises: *decimal*, *hexadecimal*, and *octal*. The decimal, or base 10, integer is the most familiar, and appears as you would expect. The important thing to note about the decimal integer is that it does not have a leading zero. The hexadecimal, or base 16, integer is typically used as a binary shorthand, each digit grouping four binary ones and zeros. Hexadecimal integers are represented by the digits 0–9 and the upper- or lowercase letters A–F, which represent the numbers 10–15. These integers are preceded by 0x or 0X. Octal, or base 8, integers are represented by the presence of a zero (0) in front of the digits. Table 3.2 gives examples of several numbers in the different formats.

Table 3.2 *Decimal, Octal, and Hexadecimal Representations of Integers*

Decimal	Octal	Hexadecimal
0	0	0x0
2	02	0x2
63	077	0x3f
83	0123	0x53
631	0771	0x3Fl

The numbers used earlier in the equation for the `fahrToCels` function (found in the previous "Comments" section) are examples of using integer literals in a script. In this case, the integer literals are used to convert a Fahrenheit temperature to Celsius.

```
CelsTemp = (fahrTemp-32)*5/9
```

Floating-Point Literals

Floating-point literals represent decimal numbers with fractional parts such as 1.5 or 43.7. They can be in either standard or scientific notation. Here are some examples:

```
3.1415, 0.1, .3, 6.022e23, 2.997E8, 1.602e-19
```

If, in the temperature conversion, you didn't want to first multiply by five and then divide by nine, you could use a floating-point number and just multiply by a single number:

```
celsTemp = (fahrTemp - 32)*0.5555556
```

Although carrying out only one operation instead of two appears to be more efficient, try typing the following code and replacing the line that does the calculation with the two previous examples.

```
<HTML>
<HEAD>
<TITLE>Say Hello:</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--  this line hides the script from old browsers
    /**
     * function to convert a Fahrenheit
     * temperature to Celsius
     */
    function fahrToCels(fahrTemp) {
        celsTemp = (fahrTemp-32)*5/9    // this does the conversion
        return celsTemp                 // this returns the Celsius
➥number
    }
        document.write("The boiling point of water in Celsius is: ")
        document.write(fahrToCels(212))
// this is the end of the script and comment structure-->
</SCRIPT>
</HEAD>
<BODY>
<HR>
</BODY>
</HTML>
```

When you run both of these, what do you notice? The resulting value for the equation using the floating-point number should not be exactly 100. In this case, there is less precision because we had to round off the floating-point number instead of using an infinite number of fives. On many computers, the storage and calculation of integers is much more efficient than with floating-point numbers. It is often better general practice to represent expressions using integers instead of floating-point numbers.

Boolean Literals

The boolean literal has two states—true and false—which are appropriately represented by the keywords `true` and `false`. The boolean literal, therefore, represents the state of something that can have only one of two values. The values are typically used as checkpoints for determining whether to take a certain action. The boolean value is a true literal, and not a representation of the integers zero or one as in C or C++. The following code shows how you use boolean literals.

```
flag = true
if(flag) {
    document.write("This line will print")
    flag = false
}
if(flag) {
    document.write('This line will not print')
}
```

The `if` statement is covered in greater depth in the next chapter. For now, all you need to know is that the `if` statement checks to see whether the statement in the parentheses is true or false. If the statement is true, it executes the statements between the curly braces.

String Literals

The term *string literal* refers to any number of characters enclosed in double (") or single (') quotation marks. Table 3.3 gives examples of some strings and their printed output.

Table 3.3 *Examples of Valid Strings*

The Declaration	The Result
`""`	
`"\'"`	'
`"Your Ad Here"`	Your Ad Here
`"Multiple\nLines"`	Multiple Lines

If you would like to see what some personalized strings look like on-screen, you can take our Hello, World! script and change the string in the `document.write` command.

```
document.write("place your String here")
```

Reload the HTML page from the Hello, World! example after you have changed the string, and see how the result has changed.

The backslash (\) is used to represent nonprinting or conflicting characters. Table 3.4 lists the various nonprinting control character combinations JavaScript accepts. These characters provide formatting of characters that are not provided on a standard keyboard, and so must be implemented with a control code. They can be used in both string literals and variables.

Table 3.4 *Special Character Representations*

Description	Standard Designation	Sequence
Continuation	<newline>	\
New line	NL(LF)	\n
Horizontal tab	HT	\t
Backspace	BS	\b
Carriage return	CR	\r
Form feed	FF	\f
Backslash	\	\\
Single quotation mark	'	\'
Double quotation mark	"	\"

Identifiers

Identifiers are the names given to variables and functions to identify them to the interpreter. Identifiers used in previous examples of JavaScript code include `helloWorld`, `text`, and `number`. What, then, makes a valid identifier? In JavaScript, all identifiers must begin with a letter or the underscore character (_). All subsequent characters also can include digits (0–9). Letters are considered the upper- and lowercase alphabet from A to Z.

Also, the words designated as *keywords* in the following section are unusable as identifiers. Table 3.5 gives examples of valid and invalid identifiers in Java.

Table 3.5 *Valid Identifiers*

Valid	Invalid	Valid, but Not Recommended
`watts`	`wattage #`	`WATTS`
`lightOn`	`light-on`	`lighton`
`monthsWith_31_days`	`5dogs`	`_number`
`x`	`abstract`	`$_243_fubar`

In the case of the invalid identifiers above, the following rules apply:

- □ `wattage #`. There can be no white spaces within an identifier.
- □ `light-on`. The hyphen (-) is an invalid character.
- □ `5dogs`. Cannot use a number to start an identifier.
- □ `abstract`. Abstract is a keyword.

Of course, unless you have a non-American standard keyboard, using anything other than the _, and A–Z letters would merely create difficulties in editing, so you shouldn't have any problems following those standards. Unless you have good reason, it is advisable to only use the _ in the middle of identifiers (between words) to improve readability. This practice is most useful to Java programmers, but getting into the right habit now will help you make the transition later. Using descriptive names should provide you with all the flexibility you need. As a rule of thumb, when you create identifiers, make all the letters lowercase except for the beginning of words that appear in the middle of an identifier (such as in `lightOn`).

NOTE In C and C++, the standard is to name #define identifiers with all uppercase letters. JavaScript does not implement a #define, so for ease of transition, all-uppercase identifiers should not be used. This is not a requirement; however, this rule will come in handy later when you are using Java, and it is good to avoid the habit of using all uppercase.

Keywords

Keywords are identifiers used by the JavaScript language, and cannot be used in any other way than that defined by the JavaScript language. You probably won't be able to remember every single keyword, so if you are having a problem with an identifier that is a single lowercase word, be sure to refer to the keyword list in Table 3.6.

Table 3.6 *List of Reserved JavaScript Keywords*

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

* Reserved keywords, but currently unused by JavaScript

Separators

In JavaScript, the spaces, tabs, and new lines between characters are known as *separators*. These are essentially removed by the interpreter; therefore, how you use them is primarily a matter for your own aesthetic sense. For example, the following code fragments are all spaced differently, but they are all equally valid expressions.

```
celsTemp = (fahrTemp - 32) * 5 / 9

celsTemp=(fahrTemp-32)*5/9

celsTemp    =    (fahrTemp-32)*5/9
```

How you choose to format your code is up to you. It is easier to read the first example with spaces between everything, but sometimes, as in the second version, you will want to make sure everything fits on one line. If you are listing items, it is sometimes useful to line up statements over several lines, so inserting spaces helps.

```
name    = "Hal Hanford"
address = "401 Rainier Way"
city    = "Black Diamond"
state   = "Washington"
zip     = 98504
```

Whatever the situation, remember that separators help to make code easier to *read,* not to run. Strive for a visually pleasing layout that will make sense to other programmers.

Declaring Variables

Variables are used to store data values in named containers that can then be referenced later. For example, in the `fahrToCels` function used in previous examples, the `fahrTemp` and `celsTemp` were variables that held the Fahrenheit and Celsius temperatures, respectively. One of the first things you do in most programming languages is declare variables based upon what type of data you want them to hold. This is typically done for more efficient code compilation and also in languages that enforce strong type checking. *Strong* type checking means the compiler or interpreter checks to make sure that each variable declares what type of data (for example, integer or string) it can contain and then makes that variable continue to contain only data of that type. Because JavaScript uses a *loose* type-checking architecture, there is no need to declare what type of value each variable will hold. Loose type checking means that a variable can hold any type of data without the need to declare a specific type. If you move on to Java, which is a strongly typed language, you will be required to properly declare variables to avoid all sorts of unpleasantness at compile time.

There are two main reasons for implementing a strongly typed language: storage optimization and robustness. (There's that "robust" word again.) If the compiler knows beforehand what type of data a variable can hold, then it can set aside enough memory and not worry about it anymore. If it doesn't know and has to constantly adjust the amount of memory a variable has, extra processing cycles are used to make adjustments, which causes the program to run more slowly. To maintain run-time speed, the compiler must be able to detect when the programmer is trying to fit data into a variable that isn't allocated sufficient memory to accept the data. Otherwise, the program could lose valuable information. In JavaScript, a variable can be adjusted automatically to make room for the larger data size. In Java, data that doesn't fit is clipped to fit, and, thus, rendered inaccurate.

For now, all you need to do is the following:

1. Use a valid name, as defined by the section on identifiers in the previous section.
2. Be sure not to use a JavaScript keyword as an identifier.

If you are familiar with programming and of declaring variables before you use them, keep this notion in mind because you will find it useful later. For now, however, you can create variables without any type declaration. The act of assigning them a value is what declares them in JavaScript. You use the equal (=) sign to assign a value to a variable, just as you have done in previous examples.

```
number = 10

name = "Steve"

cost = 10.99

isLightOn = false
```

NOTE One good piece of advice: try to declare all variables and assign them a value at the beginning of code blocks where they are used. This makes the function easier to read, because right away you can see all the variables the function will be using.

It also is a good habit to make sure that all your variables exist in a known state before program execution enters a complex section of code. This means that if you have to locate a bug, you at least know all of the values when entering the troublesome section. When you move on to a stricter, object-oriented programming language like Java, this advice becomes even more useful.

Using var for Variables

Of course, now that you have been told to go create variables wherever it pleases you, it's time to bring some order back to things; snatching control from the jaws of anarchy is a recurring theme in this book. One problem with the dynamic interpretive nature of JavaScript is that it is difficult to catch errors in code, especially in duplicate variable names. For example, you may bring in a JavaScript source file using the SRC attribute that uses the same variable name as a function you are embedding directly in the HTML page. Which one takes precedence? The answer is clear cut: *if* you use the keyword var inside the block of code where the variable is created. With a var preceding a variables first use in a block, this variable will be used in all expressions inside this block instead of any other global variables that have the same name.

The following code shows how the var statement can affect the execution of a script. Run this, and see what happens:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript" >
<!-- hide script from old browsers

function test() {
     flag = true

     document.write(flag + " ")
     test1()
     document.write(flag + " ")
     test2()
     document.write(flag + " ")
}

function test1() {
     var flag = false
     document.write(flag + " ")
}

function test2() {
     flag = false
     document.write(flag + " ")
}

// end script hiding -->
</SCRIPT>
<BODY>
<SCRIPT LANGUAGE="JavaScript" >
<!-- hide script from old browsers
test()
// end script hiding -->
</SCRIPT>
</BODY>
</HTML>
```

If you run this in Netscape 2.0, you will notice that the output comes out as

```
true false true false false
```

when what you might expect is either this:

```
true false false false false
```

... or this:

```
true false true false true
```

The two functions that are used by the first function happen to use the same name for `flag`. The function `test1` is a friendly function and uses the `var` statement to ensure that it doesn't affect any global variable of the same name in existing code. Function `test2`, on the other hand, declares the variable `flag` without regard to any other code that might be trying to use the same name and overrides the calling function's value for `flag`. Although this is a simple example, you don't always know where faulty code comes from, especially if there are several programmers sharing a project. Knowing about variable precedence can help you isolate problems like this.

A similar situation could happen if some functions you needed were loaded using the SRC attribute, to which you then added your own code. You might know how to call the functions you imported, but you might not know what the guts of the scripts were. If the programmer didn't use `var`, and there was a variable that matched, you could have a function alter variables in ways that could be difficult to track down.

For this reason, whenever you create a variable, be sure to use `var` in front of it when you want to ensure it doesn't replace values in any global variables that might have the same name.

Creating Expressions and Using Operators

After you have created your variables, you must be able to assign them values, make changes to them, and perform calculations. These are the roles of the operators. Table 3.7 lists the operator precedence from highest to lowest.

Table 3.7 *Operator Precedence from Highest to Lowest*

. [] ()
++ -- ! ~
* / %
+ -
<< >> >>>
< > <= >=
== !=
&
^
¦
&&
¦¦
?:
= += -= *= /= %= <<= >>=>>>= &= ^= ¦=
,

Precedence refers to the order in which multiple operations are computed. Operators on the same level have equal precedence. For example, the following calculation

```
a=b+c*d/(c^d)
```

... proceeds by working from left to right on all binary operations (those involving two variables), computing those operations at the top of the list and working downward. In this case, because the () has higher precedence than anything else, the c^d is computed first. Next, the c*d would be computed and that result divided by the result of the first operation. Finally, everything is added to b and the result placed in a. Remind you of high school algebra at all?

NOTE Whenever you are in doubt about the order in which something is calculated, be sure to use the parentheses () to tell the compiler which terms in the expression to compute first.

The first operator you need to know is the assignment operator (=). The assignment operator takes the values on the right side of the equal sign and places them into the variable on the left side of the equal sign. Even though this is the easiest operator, it can get you into lots of trouble. Later, you will look at the == operator, which compares two values, ***but does not change either one.*** Instead, it returns whether they were equal at the time of the operation. It does not alter the left-hand operant to equal the right-hand operand, as the = operator does.

Casting Variables

JavaScript attempts to convert data from one type to another type when needed, but sometimes it is unable to do this. This is a problem unique to a dynamic language like JavaScript. Because there is no compiler to check whether assignments are correct before they run, the programmer must be careful when writing scripts.

One example for which JavaScript might not convert is when trying to make a string into a number. As long as the string ***represents*** a number already, the conversion works. If the string represents a series of letters, however, the conversion will fail with an error. The following table (3.8) lists the results of almost all conversions.

Table 3.8 *Conversions Between Types*

Data Type	*Target Data Type*				
	function	object	number	boolean	string
function		function	error	error	decompile
object	error		error	true	to String
Null object	funobj OK		0	false	"null"

continues

Table 3.8 *Continued*

Data Type	*Target Data Type*				
	function	object	number	boolean	string
number (nonzero)	error	number		true	to String
number (0)	error	null		false	"0"
Error (NaN)	error	number		false	"NaN"
+ infinity	error	number		true	"+Infinity"
- infinity	error	number		true	"-Infinity"
false	error	boolean	0		"false"
true	error	boolean	1		"true"
string (nonnull)	funstr OK	string	numstr OK	true	
null string	error	string	error	false	

Some of these results might not make sense to you now; that's okay. As you learn more of the language and begin to use objects and functions, come back to this list and compare how different objects convert among types to make sure you understand what is going on in examples where conversions take place.

Arithmetic Expressions

Now that you have learned how to create variables, it's time to learn how to use them. JavaScript has several operators that can be used on variables—some of which are specific to certain types of variables and return specific kinds of values. Some of these values we saw in Table 3.8. There are three types of operators: arithmetic, logical, and string. The last three sections of this chapter present these different operators and discuss their common usage.

Additionally, arithmetic operators come in two flavors: binary and unary. *Binary* operators require two variables to work on, while *unary* operate on a single variable. Table 3.9 lists the unary operators.

Table 3.9 *Unary Integer Operators*

Operator	Operation
.	Unary negation
-	Bitwise complement
++	Increment
--	Decrement

The *unary negation* changes the sign of a number. *Bitwise complement* changes each bit of the variable to 1 if it is a 0, and to 0 if it is a 1. This operator is useful if you want to twiddle with the underlying binary representation of a variable and is a practice most common in advanced languages like C and C++. For example, the character "A" is represented by the ASCII code 65, which is 1000001 in binary notation. A bitwise complement operator could be used to alter the binary value of "A" and make it "Z." *Increment* increases the value of the variable by one, and *decrement* decreases the value of the variable by one. Here is an example:

```
i = 0
j = 10
for(i = 0; i<10; i++) {
     j--
     document.write(i+"\t"+j)
     document.write("<p>")
}
```

This script prints increasing numbers in one column and decreasing numbers in the other. Note the use of ++ and --. Each time these occur, the system either raises or lowers the value of the operand by one. This is the way unary operators work—they change the value of the variable they are used on. For the negation and bitwise complement, the variable is not changed; for the increment and decrement, the variable is changed. The following code gives an example of the way this works:

```
i = 10, j = 10, k = 10, l = 10
     document.write(i+"\t"+j + "\t" + k + "\t" + l)
     document.write("<p>")
j++
i--
~k
-l
document.write(i+"\t"+j+"\t"+k+"\t"+l)
document.write("<p>")
```

Notice that j and i have been changed and print out their new values, but k and l have reverted to their original values. When you use the unary negation and bitwise complement in a compound operation, you actually use a temporary variable that holds the new value of the operand. The increment and decrement operators are both prefix and postfix—that is, they can be placed before (++x) or after (x++) the operand. If they are used in compound statements such as

```
i=x++
```

or

```
i=++x
```

then the first line increments x *after* assigning its value to i, and the second line increments x and then passes the new value on to i.

The second type of integer operator is the binary operator. These operations do not change the values of the operands. They return a value that must be assigned to a variable. Table 3.10 lists the binary integer operators.

Table 3.10 *Binary Integer Operators*

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
¦	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift
>>>	Zero-fill right shift

The following program prints the values of some operations:

```
i = 5
j = 10
document.write(i+"\t"+j)
document.write("<p>")
j = j + i
document.write(i+"\t"+j)
document.write("<p>")
j = j * i
document.write(i+"\t"+j)
document.write("<p>")
j -= i
document.write(i+"\t"+j)
```

Notice the last operation. It is a combination of the binary operator and the assignment operator. This is equivalent to writing `j=j-i`. This can be done with all the binary operators, and is a common shorthand to get used to. If the operator is

```
x [op]= y
```

then the expression is equivalent to

```
x = x [op] y
```

Take the time to place your own equations into the code. Add variables and try different combinations.

NOTE Here are some further notes on the integer operations. First, division of integers rounds toward zero. Second, if you divide or modulo by zero, you will have an exception error at run time. If your operation exceeds the lower limit, or underflows, the result will be a zero. If it exceeds the upper limit, or overflows, it will lead to wrap-around. Moving past the upper limit will place you at the very bottom value—approximately -2.1 billion.

There also are additional relational operators that produce boolean results. These operators are shown in Table 3.11.

Table 3.11 *Relational Integer Operators that Produce Boolean Results*

Operator	Operation
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

The equal-to operator (==) can cause you endless suffering. I still sometimes replace the double equal sign with just a single equal sign when I mean to compare two values instead of assigning the value of the right operand to the left. Make sure that you use the double equal sign for comparison. Try this application:

```
i = 0
j = 10
while(i<j) {
    document.write(i)
    i++
}
```

Here, the control statement `while` checks the values of `i` and `j`. While the statement `i<j` is true, it runs the code fragment that prints `i` and then increases it by one. As soon as `i<j` is false, the program drops out of the `while` loop and finishes. We will cover `while` loops in Chapter 4.

Logical Operators on Boolean Types

The boolean type adds several new operators for logical computation. These operators are listed in Table 3.12.

Table 3.12 *Boolean Operators*

Operator	Operation
`!`	Boolean negation
`&`	Logical AND
`¦`	Logical OR
`^`	Logical XOR
`&&`	Evaluation AND
`¦¦`	Evaluation OR
`==`	Equal to
`!=`	Not equal to
`&=`	AND assignment
`¦=`	OR assignment
`^=`	XOR assignment
`?:`	Ternary

If you consider the boolean value to be the equivalent of either 1 for true or 0 for false, the operators act the same as those for the integer operators if they are working on a single bit. Negation (`!`) is the equivalent of the integer bitwise complement (`-`) and is a unary operation. Table 3.13 lists the results of the operations.

Table 3.13 *Results of Boolean Operations*

AND			*OR*			*XOR*		
Op1	Op2	Result	Op1	Op2	Result	Op1	Op2	Result
true	true	true	true	true	true	true	true	false
true	false	false	true	false	true	true	false	true
false	true	false	false	true	true	false	true	true
false	false	false	false	false	false	false	false	false

The `&`, `¦`, and `^` operators evaluate both sides of the argument before deciding on the result. The `&&` and `¦¦` operators can be used to avoid evaluation of right-hand operands if the evaluation is not needed.

The `?:`, or ternary, operator works as shown here:

```
Operand1 ? Statement1 : Statement2
```

`Operand1` is evaluated for its truth or falsity. If it is true, `Statement1` is completed; if it is false, `Statement2` is completed. The following code provides an example of this operation:

```
i = 10
j = 10
test = false
test ? (i = 0) : (j = 0)
document.write(i+"\t"+j)
test = true
test ? (i = 0): (j = 0)
document.write(i+"\t"+j)
```

Note the parentheses. They are not needed in this example, but if you use more complicated statements, you might want to include them.

NOTE

In addition to these operators, the `&=`, `¦=`, and `^=` work as assignment operators on boolean values just as they do for numerical values.

Operators on Floating-Point Numbers

The traditional binary operations work on floating-point values (-, +, *, and /) as well as the assignment operators (+=, -=, *=, and /=). Modulus (%) and the modulus assignment operator (%=) are the floating-point equivalent of an integer divide. Also, the increment and decrement (++, --) increase or decrease the value of the variable by 1.0.

Operators on Strings

Strings can be concatenated using the + operator. If any of the operands are not strings, they are converted to strings before being concatenated. In addition, the += operator works by placing the concatenation of two strings into the left-hand operand. You used the + operator in the previous examples when you wanted to print several items on one line. Try using the `document.write()` function and the + operator to make different combinations of output with the interpreter.

NOTE **The left-hand operand in a += string operation must already have a value in order to work in Netscape 2.0. For example:**

```
string2 = "Hello"
string1 += string2
```

does not work because `string1` is not defined. However:

```
string1 = ""
string2 = "Hello"
string1 += string2
```

does work.

As far as using JavaScript for computation goes, you have now covered all the basic operations that can be performed on variables. You should spend time with the information presented and experiment with your own programs. Of course, even in the examples here, you used several control flow statements to make the programs really do something. Control flow provides logic within your programs and the basic engine that makes your calculations work.

In the next chapter, you learn the control flows available in JavaScript.

Summary

This chapter covered the basic architectural components, or tokens, of the JavaScript language. These components included correct JavaScript code placement in HTML pages, keywords, variable declaration, and issues of general syntax. This chapter also discussed the operators that can be used to build up the computational aspects of a JavaScript program. As I hinted earlier, the material in this chapter provides building blocks for JavaScript applications along with the set of parts that is available when you create additional complexity later on. There were also several tips for good code formatting and commenting.

In the next chapter, you will delve into the organizational and control flow aspects of the language where you can really begin creating some full-blown JavaScripts. The importance of these methods has already been shown in some of these early examples. It is difficult, for example, to do anything beyond saying "hello world" without control flow (such as `while` and `if…else`), and I assume you have grander ambitions for JavaScript.

After you have mastered the concept of control flow, it will be time to get into functions and event handlers, where lies much of JavaScript's exciting functionality. You will handle events from user input controls on HTML forms and create functions that react to this input without resorting to CGI scripts.

CHAPTER 4

Control Flow and Functions in JavaScript

So far, the JavaScript language fundamentals have been covered, but there is not much more you can do with these elements except somewhat trivial applications. In order to really add functionality to your program, you need to begin using control flow statements and functions. These enable you to create complex actions that respond to computational results in the script or input from the system or user. *Control flow statements* enable programs to pick execution paths depending on certain conditions you define.

Control flow statements are the basic organizational tools used in programming languages. JavaScript provides several control flow statements that can be used to do the following:

□ Repeat processing blocks of code for specified intervals

□ Make branch decisions

□ Return control to the main script body from function calls

□ Enable the path of execution to be altered in a dynamic fashion

Functions in JavaScript can be declared as standalone routines that can be called from additional embedded scripts or from event handlers in the HTML code, such as form buttons. In this respect, JavaScript differs from Java in that functions are not required to be a part of an object, although they can be, as we will see in Chapter 6, "JavaScript and Built-In Objects."

Functions in JavaScript enable you to do the following:

- □ Encapsulate often-used code in a single routine that can be called at different points in the entire program
- □ Allow the use of event handlers in HTML pages that can create certain behavior based on user input

After you have finished this chapter, you will have all the basic tools necessary for creating useful JavaScript scripts to extend your capabilities well beyond "Hello, World!". In the next chapter, "Using and Creating Objects in JavaScript," you will look at the areas in which JavaScript really excels—the use of objects provided by the Netscape environment and created from scratch in JavaScript code. It is this object-based capability that enables JavaScript to access exposed methods and properties of objects, such as document, window, location, and history, along with built-in JavaScript objects for utilities such as date and math functions. This same capability is used to access plug-ins and Java applets in order to script together diverse components that help you create truly object-oriented Web pages.

Control Flow in JavaScript

Control flow is a set of methods used to make program execution move through the code in an order you specify. JavaScript provides several control flow expressions, as shown in Table 4.1. Control flow is the heart of a JavaScript program, providing it with capability you specify to make decisions. Rarely does a program go through a single, linear series of steps to produce its output. To enable interaction, programs must be able to react to input and calculations and make decisions about what code to execute next. In addition, if you need to repeat a code fragment several times in a row, it is a waste of effort to type the same code block in each place the program will need it—instead, you can tell the JavaScript interpreter to repeat the code block. JavaScript provides control flow methods for these and other situations.

the more complex objects created in languages such as Java into a unified structure. JavaScript does this by providing an HTML-based language that can more easily be implemented by Web page developers than Java or even CGI scripts.

Although Java's object-based language architecture does not provide many of the most common features of OOP languages such as inheritance, encapsulation, and abstraction (again it's all just post-modern art criticism, unless you really *want* to learn OOP parlance), it does provide a means for creating objects that can have properties and methods, much like those found in the Java class structure. The benefit of using objects in a JavaScript program is that after you create the object, you can reuse it in much the same way as you are able to reuse the functions created in the previous chapter. Besides creating objects, you can also use the object structure to create the JavaScript equivalent of an array. In addition to objects that you can create yourself, JavaScript and the Netscape environment provide several objects that can be used in your JavaScript programs. Chapters 6 and 7 cover these objects in more detail. For now, this chapter covers objects in two respects:

- □ Using objects that already exist
- □ Creating your own objects

After you have learned how to interact with existing objects and create your own, you have learned JavaScript's object-based fundamentals. The next chapters merely provide you with information about built-in functionality that you can use in your JavaScript programs.

Using Objects in JavaScript

To use objects in JavaScript, you must understand their basic structure as well as the elements with which you need to interact to "utilize" the functionality of the objects. In essence, there are two types of elements that make up a JavaScript object:

- □ *Properties.* A JavaScript object property is the same thing as a JavaScript variable. It is a holder for information that the object needs in order to perform its required behavior.

CHAPTER 5

Using and Creating Objects in JavaScript

One of the most important features of the JavaScript language is its capability to create and use objects. These objects enable you to implement powerful models for application development. Use of these models not only eases design and implementation details of complex programs, but also enables you to bring together previously created objects for use in current projects.

JavaScript's implementation of objects is not considered fully object-oriented, because it does not provide the basic properties of object-oriented languages, such as abstraction, inheritance, and encapsulation. If these sound like terms best used to describe post-modern artistic criticism, don't worry; you won't need to understand them to use JavaScript. Simply bear in mind that whatever your background in object-oriented programming, JavaScript is not strictly object-oriented, and so many of the conceptual headaches that go along with objects won't be your burden to bear. JavaScript is not able to provide the means to create objects like those available in the Java language. Its capability, however, to *use* objects that have been created in other more traditional *object-oriented programming* (OOP) languages such as Java has led JavaScript designers to call JavaScript an object-based language. For JavaScript to have implemented the full object-oriented paradigm would have been considered overkill and would have missed the main point of the language: to script simple programs for execution in HTML pages that can tie together objects that exist in the browser environment, such as Java applets. A fully object-oriented programming language already exists in Java. It is JavaScript's job to pull together available in the Netscape for Macintosh environment. Of all the chapters in the book, the next one is the most important. Be sure that you understand all of the basic points before continuing, because this basic information is required in order to develop powerful, object-based scripts for your HTML pages.

```
<FORM>
<INPUT TYPE="button" VALUE="Test" onClick="i++; buttonPressed =
➥'Test'">
</FORM>
```

The use of a single function, however, makes this code much more modular by enabling changes in the code to be made in one place, not to mention the fact that a single function call makes the event handler easier to read.

The next chapter provides a more in-depth discussion of event handlers, and presents form objects and examples of the events they handle. For now, it is important to see how functions can be used in relation to the event handler, and to get a feel for the overall usability of the function declaration.

NOTE Later, as you begin to look at the objects that are programmable by the JavaScript language, such as form elements, you will access the event handler of these elements directly. Also, HTML enables you to set the event handler by calling `onCLick`, `ONCLICK`, or `onclick`—it's case insensitive. Because of the way the JavaScript symbols are stored, if you want to refer to an event handler by name, you must use all lowercase letters, such as `onclick`.

Summary

The control flow and function statements are simple yet powerful means for creating smooth-flowing, modular, easy-to-maintain scripts for HTML pages. These features of the language are used time and again in the first half of this book, and many of them are used in Java. In fact, almost all of the programming examples include one of these items to enable complex behavior that is worth exploring.

In the next chapter, the JavaScript basics will be moved aside for more complicated matters. It is time to look at objects: how to create them in JavaScript, how to use them, and what objects are

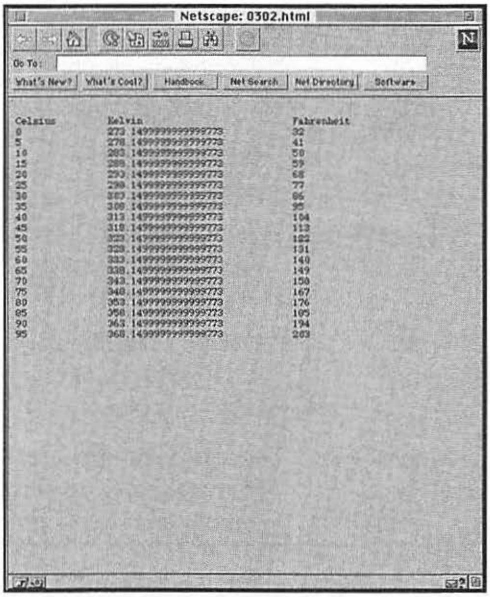
Table 4.2 *Continued*

Event	Event Description	Object
Change	Occurs when data is changed or script action	Text fields, text areas, selections
Click	Occurs when the user clicks the mouse button while the mouse pointer is over one of these objects	Buttons, radio buttons, checkboxes, submit buttons, reset buttons, links
Select	Occurs when the user selects a block of text within an object	Text fields, text areas
MouseOver	Occurs when the user moves the mouse pointer directly over a link object	Links

In order to process an event for a field, an event handler attribute must be added to the tag for the field. For example, you can cause a button to invoke a dialog box using the following tag:

```
<FORM>
<INPUT TYPE="button" VALUE="Test" onClick="alert('This is a test')">
</FORM>
```

The `onClick` attribute tells the interpreter what JavaScript code to run when the designated event occurs. In this example, the alert function is built into the JavaScript language, but you can also substitute new functions. One item to note is the use of single quotes in order to demarcate HTML tag attributes from function arguments. The limitations of text in HTML dictate that quotations being sent to JavaScript functions or variables are of a different quote style than that used to indicate what is to be executed for the event handler. For this reason, if you use `onClick=" ... "`, you must use single quotes inside, as in the following code fragment. If you use `onClick=' ... '`, you must use double quotes for any statements inside. Of course, you don't have to call a function. You can also place plain JavaScript code in this attribute tag, separating statements using a semicolon (`;`):



```
Celsius     Kelvin                     Fahrenheit
0           273.14999999999999773      32
5           278.14999999999999773      41
10          283.14999999999999773      50
15          288.14999999999999773      59
20          293.14999999999999773      68
25          298.14999999999999773      77
30          303.14999999999999773      86
35          308.14999999999999773      95
40          313.14999999999999773      104
45          318.14999999999999773      113
50          323.14999999999999773      122
55          328.14999999999999773      131
60          333.14999999999999773      140
65          338.14999999999999773      149
70          343.14999999999999773      158
75          348.14999999999999773      167
80          353.14999999999999773      176
85          358.14999999999999773      185
90          363.14999999999999773      194
95          368.14999999999999773      203
```

Figure 4.2 *The* `tempConvert()` *method in action.*

Triggering Functions Using Event Handlers

Of course, there is another way in which functions can be used, and that is as an event handler for objects. The use of forms and handling events will be discussed in more detail in Chapter 8, but because you have already used event handlers before in the example code that used form buttons, it is worthwhile mentioning them here.

Table 4.2 *Events That JavaScript Can Handle*

Event	Event Description	Object
Focus	Occurs when an object is set for data entry by a user or by a script	Mouse clicks, tab, text fields, keypresses, text areas, selections
Blur	Occurs when an object loses focus, that is, the user presses the tab key to move to the next text field from the current one	Text fields, text areas, selections

continues

```
    else {
        alert("Error - arguments incorrect")
        return fromTemp
    }
    return toTemp
}
// end script hiding -->
</SCRIPT>
</HEAD>
<BODY>
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!-- begin hiding of script
document.write("Celsius\t\tKelvin\t\t\t\tFahrenheit\n")
for(i=0;i<100; i+=5)
    document.write(i + "\t" + "\t" + tempConvert(i,"CK") + "\t"+ "\t"
➥+ tempConvert(i,"CF") + "<br>")
// end script hiding -->
</SCRIPT>
</PRE>
</BODY>
</HTML>
```

Figure 4.2 shows the resulting output of the JavaScript code in an HTML page. You should notice the use of the <PRE></PRE> tags before and after the <SCRIPT></SCRIPT> tag. This enables you to use text formatting characters such as tabs, spaces, and newlines in order to format the display of text on-screen without resorting to HTML tags. This is important because some non-Netscape browsers do not interpret HTML comments correctly—especially comments that contain other HTML tags. Therefore, having a > character anywhere between the <SCRIPT></SCRIPT> tags could disrupt the display of your Web page for some of these browsers. By using <PRE></PRE> tags, you can format text using standard tabs, carriage returns, and spaces without worrying about how various browsers might interpret your code.

number would necessarily be given, it can check to make sure that it has information to deal with each type of number, be it local, long distance, international, or to a local exchange.

NOTE . arguments is actually an array. Although most languages have built-in array types for holding variables, JavaScript does not, except for those provided by the language environment such as arguments and elements you will see later. For more information on arrays, see Chapter 5, "Using and Creating Objects in JavaScript."

Using Functions and Events

In order to use functions, you must call them in either a script or an event handler. Calling functions in scripts is a rather simple affair, and requires only that you call the function by using the name and including the required arguments. The tempConvert() function, for example, can be used to create a table of values for a temperature conversion table, as follows:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!-- begin hiding of script
function tempConvert(fromTemp, cType) {
    var toTemp = 0
    cType.toUpperCase()
    if(cType =="CK")
        toTemp = fromTemp + 273.15
    else if(cType == "CF")
        toTemp = fromTemp * 9 / 5 + 32
    else if(cType == "FK")
        toTemp = ((fromTemp - 32)* 5 / 9) + 273.15
    else if(cType == "FC")
        toTemp = (fromTemp - 32) * 5 / 9
    else if(cType == "KC")
        toTemp = fromTemp - 273.15
    else if(cType == "KF")
        toTemp = ((fromTemp - 273.15) * 9 / 5) + 32
```

global variables intentionally. Whatever variables you need you should either create as local vars or require that they be sent as arguments that can then be used by the function. This way, you are assured that your functions will be portable to other HTML pages, and that they can be mixed in with scripts without requiring that you know the implementation details or have to worry about how the variables might conflict.

Allowing for Variable Argument Lengths

You might want to have a function that can take several numbers of arguments. In this case, how do you know how many arguments have been passed to your function? In this case, you can use the arguments variable, which is an array that holds the values of all the arguments passed to the function. You might, for example, have a function that takes four arguments, but requires only one:

```
function setNumber(number, areaCode, countryCode, extension) {
...
}
```

The number of arguments passed is stored in the variable arguments.length. By checking the number stored in arguments.length, you will know how many have been passed to the function.

```
function setNumber(number, areaCode, countryCode, extension) {
    this.number = number;
    if(setNumber.arguments.length > 1)
        this.areaCode = areaCode;
    if(setNumber.arguments.length > 2)
        this.countryCode = countryCode;
    if(setNumber.arguments.length > 3)
        this.extension = extension;

// then do something with the number
...
}
```

In this case, the function could store the values of the phone number depending on which were given. Because only the first

In this case, newTemp would now be equal to 212. Notice that instead of requiring you to write the equation for the conversion every time, you need only to provide the necessary information, and the function returns the correct value. In addition to sending back the correct answer, if the conversion type is unknown, then an alert dialog box opens up and the original temperature is returned to the calling statement.

Using var in Functions

Recall from our earlier discussion on the declaration of variables that the act of assigning a variable a value is what declares it in JavaScript. That's it. No DIM statement or other arcane syntax is necessary to tell JavaScript what to do with a variable. You then use the equal (=) sign to assign a value to a variable, just as you have seen in previous examples. For example,

```
wattage=100
```

simultaneously declares the variable wattage and sets its value to 100.

In the temperature conversion function, you may have noticed the use of the var statement in front of the variable declaration for toTemp. You can use var statement when you want to make sure that the variable is completely local. In the case of the toTemp variable, you don't know if that variable has already been used somewhere else. Because you don't want to inadvertently change data in a global variable, you should make sure that all variables declared and used in a function are declared var to ensure protection of possible variables sharing the same name.

NOTE **Using var— is a good habit to get into and increases the safety and reliability of your code.**

One related issue: In addition to ensuring that you don't inadvertently alter global variables by using the var statement, it is also important from a design standpoint that you refrain from altering

```
    else if(cType == "CF")
        toTemp = fromTemp * 9 / 5 + 32
    else if(cType == "FK")
        toTemp = ((fromTemp - 32)* 5 / 9) + 273.15
    else if(cType == "FC")
        toTemp = (fromTemp - 32) * 5 / 9
    else if(cType == "KC")
        toTemp = fromTemp - 273.15
    else if(cType == "KF")
        toTemp = ((fromTemp - 273.15) * 9 / 5) + 32
    else {
        alert("Error - arguments incorrect")
        return fromTemp
    }
    return toTemp
}
```

In this version of the conversion function, it takes a temperature, which can either be an integer or a floating-point number, and a two-character string, which indicates what scales to convert from and to, respectively. For example, "CK" indicates the function should convert the `fromTemp` argument *from* Celsius *to* Kelvin. Then, using this information, the conversion is made and returned to the calling statement.

NOTE A note of advice. It is optimal if you can place all your function declarations inside the <HEAD></HEAD> tag, because this is read and interpreted before the rest of the Web page. This makes sure that any functions that might be called in the main HTML document will have been loaded into memory.

Using the return Statement in a Function

You might have noticed two statements that use the `return` keyword. These statements send values back to the statement that called the function. For example, say you wanted to convert the temperature of 100 degrees Celsius to its equivalent in Fahrenheit. You might call the function in a JavaScript later in the page by invoking the function as follows:

```
newTemp = tempConvert(100,"CF")
```

function as it pertains to the extension of HTML capabilities. You have already seen examples of functions used as arguments for event handlers (the "game show buttons" example in Figure 4.1), so you should be familiar with this idea. In the next section, however, you go through a step-by-step process of creating functions and using them as event handlers.

In order to create a function in JavaScript, you must declare it by using the function statement:

```
function functionName(arguments) {
...
}
```

This statement creates a function that can be called using the following statement:

```
functionName(arguments)
```

For example, say you want a function that will convert between Fahrenheit, Celsius, and Kelvin. The first thing you need to ask yourself is what this function needs in order to perform this calculation. In this case, it needs to know what the temperature to convert is, what scale that temperature is in, and what scale to convert to. Therefore, your function needs to take three arguments: temp, fromScale, and toScale. The function statement would look like the following code:

```
function tempConvert(temp, cType) {
...
}
```

This defines the function. After it has been established what the function needs to do and what arguments it takes, the body of the function can be created as follows:

```
function tempConvert(fromTemp, cType) {
    var toTemp = 0
    cType.toUpperCase()
    if(cType =="CK")
        toTemp = fromTemp + 273.15
    result[i] = numerator/denominator
    // update numbers
}
```

Both of these statements can be used with the for and while loops to either break out of the loop completely when it might otherwise fail, or skip the remaining code in the loop block and continue with the rest of the loop.

WARNING Make sure you attempt to catch all problems (such as divide by zero) before they appear for your site's users. Because these programs will be running on remote systems, it is important that they be as robust as possible. In Java, a more complex system for catching exceptions is used, but this system is not available in JavaScript. Getting into the habit of looking for such problems now can get you in a good position for handling exceptions later.

Creating Functions in JavaScript

The use of functions in JavaScript serves several purposes. JavaScript functions can be used to organize blocks of code that recur several times in a JavaScript program. By placing this code in a function, the programmer can encapsulate behavior and use a single command to invoke complex actions without the repetition of typing in long code every time it is needed. In addition to the greater readability and ease in programming that functions create, they also help in the actual programming of the code inside the functions—the programmer can concentrate on the actions required by the function code in isolation from the rest of the program. This modular programming idiom is the basis for much of the increased functionality of object-oriented programming.

JavaScript functions also are the basis for creating *event handlers*, which enable a JavaScript-enhanced Web page to react to user events without relying on server-side scripting and network calls. This is one of the most important requirements of the JavaScript

Typically, you will need to check for some external change in the state of a variable, such as intruderInHouse, in order to reevaluate each conditional variable that would need to change. The important thing with while statements is ensuring that you can actually get out of their loops, meaning that the code you execute in the while block must eventually change the state of the while expression itself.

Using break and continue to Control Loops

The break statement is used to exit a looping statement, such as while() or for(), immediately. The break statement continues execution at the first statement past the closing brace. You might want to use a break statement to test for a possible error condition before a computation. The following code checks to make sure the denominator in a division operation is not zero before proceeding:

```
for(i=0;i<20;i++) {
    if(denominator == 0) break
    result[i] = numerator/denominator
    // update numbers
}
```

In this case, the for loop runs until either i is 19 or the denominator equals zero. Because you don't want to divide by zero, you have the loop break so that it doesn't cause an error.

The continue statement is used when you don't want to execute any of the statements from that point to the end of the block, but want the loop to carry on with the next iteration. In the case of the previous example, if a denominator is zero, then the loop would break and the rest of the array would not be filled. Instead, you can use continue, which not only skips the actual process of updating the array, but skips past the calculation that would generate an error. The following code allows the for loop to continue, but skips any divisions where the denominator is zero:

```
for(i=0;i<20;i++) {
    if(denominator == 0) continue
```

WARNING Be careful if you use the variable that is being used as the counter in the `for` loop (such as `cels` in the previous example) within the actual block of code. If you change the number somewhere within the `for` statement block, it could alter the number of times the loop executes. Also, this might not occur in the most obvious manner, so tracking down the bug could be difficult.

You can use `for` loops in many places—for accessing array elements by index, anywhere a series of numbers is needed, or anytime something has to execute a specific number of times, such as a time delay. You will understand more about why arrays are useful when you learn about them in the next chapter.

Using Conditional `while` Loops

```
while(condition) statement
```

The statement is identical in format to the `for` and `if-else` loops in that either single statements or multiple statements surrounded in a block can be used. You can mimic the behavior of a `for` loop using `while` as can be seen in the following code.

```
i = 0
while(i < 10) {
    i++
    document.write("loop " + i)
}
```

The `for` loop is more compact, however, and makes for easier reading when dealing with a series of numbers. The `while` loop is better equipped for establishing the condition of more complicated statements, such as the following:

```
while(intruderInHouse && !policePresent) {
    soundAlarm()
    intruderInHouse = checkForIntruder()
    policePresent = checkForPolice()
}
```

- □ The last expression tells the loop what to do to the counter each time, and is known as the ***update expression***. In this example, the `i++` tells the JavaScript interpreter to increase the value of `i` by 1 each time the loop is run. This causes `i` to go from 0 to 9 through all of the intervening integers. If you've programmed in BASIC before, then ***update*** should seem familiar because it does what the `step` statement does for BASIC `for-next` loops.

You could use a statement like the following to address the members of an array in order. (Arrays are covered in the next chapter.):

```
for( i = 0 ; i < 10 ; i++) a[i] = 0
```

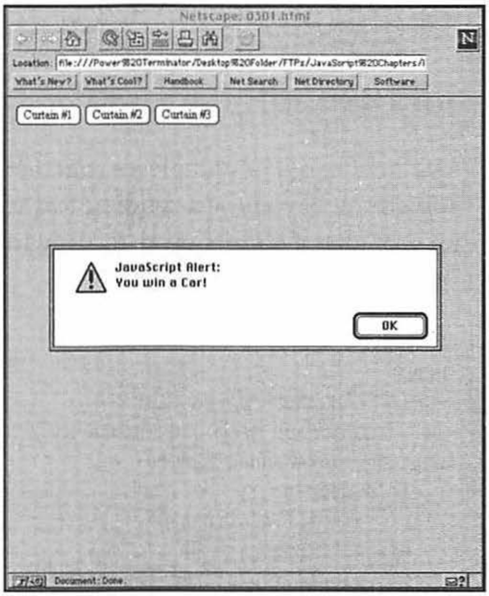
The preceding code would fill the array with integers of zero.

The last expression in the `for` statement (`i++`) can be omitted, but be sure to do something with the counter variable in your statement—if the conditional expression was true the first time, you will be in an endless loop because nothing will change that status. The following code changes the state of the `i` variable not in the `for` statement itself, but in the code block:

```
for(i = 10; i >= 0 ; ) i--
```

An example using the `for` loop to print a conversion from Celsius to Fahrenheit in five-degree increments follows:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!-- hide script from old browsers
for(cels = 0; cels <= 100; cels += 5) {
    fahr = cels * 9 / 5 + 32
    document.write(cels + " " + fahr + "<br>")
}
// end script hiding -->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```



JavaScript Alert:
You win a Car!

Figure 4.1 *The* `if-else` *control in action.*

The `for` Loop for Repeating Code

The `for` loop is a powerful tool for looping through a series of instructions until some limit has been reached. The format of the `for` statement uses a variable, often called a *loop counter* in this case, to compare with a certain limit. When the limit is reached, the loop is broken, as follows:

```
for( i=0 ; i<10 ; i++) ...
```

The format of the argument (*`ex1 ; ex2 ; ex3`*) is explained next.

- □ The first expression, `i=0`, tells the `for` loop where to start the counter variable and is known as the *initial expression*. In this case, `i` is set to `0`.

- □ The second expression, `i<10`, tells the `for` loop when it should stop and is called the *conditional expression*. In this case, the statement is true (that is, the loop should continue) for all counter values 0–9.

```
if(x > .0) {
    if(x == z) a = z
}
else a = x
```

You now have enough pieces together to write a program that does something beyond just sending text to the screen. The following is an example of a function that, depending upon the button pressed, outputs a different message:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!-- hide script from old browsers
function pickAWinner(number) {
    if(number == 1)
        alert("You win a Car!")
    else if(number == 2)
        alert("You picked the goat")
    else if(number == 3)
        alert("You get to keep your 100")
    else alert("incorrect entry")
}
// end script hiding -->
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE="button" VALUE="Curtain #1" onClick="pickAWinner(1)">
<INPUT TYPE="button" VALUE="Curtain #2" onClick="pickAWinner(2)">
<INPUT TYPE="button" VALUE="Curtain #3" onClick="pickAWinner(3)">
</FORM>
</BODY>
</HTML>
```

Obviously, the program doesn't do that much—unless you are on a game show—but you can see the way in which you can use the `if-else` statement to control the flow of the program depending on the condition of certain variables. Figure 4.1 shows the result of the JavaScript code.

If you don't want to do anything if the statement is not true, you can simply leave off the else. If you want to have your program do more than one thing for each statement, surround the statements with curly braces {}, as follows:

```
if(!lightOn) {
     wattage = 0
     roomDark = true
}

else {
     wattage = 60
     roomDark = false
}
```

In addition, if you want to string together a series of tests, you can do so as shown here:

```
if(boolean) statement
else if(boolean) statement
else if(boolean) statement
...
else statement
```

In this case, the boolean expressions at each stage are evaluated. If one is found to be true, it is executed and the rest skipped; otherwise, the final else is executed.

Be sure to use the curly braces for your statements if it is unclear in which order statements should be executed. Here's an example:

```
if(x > 0)
     if(x == z) a = z
else a = x     //this else would be associated with the inner if.
```

The preceding example produces an incorrect result if you want a to equal x only if x <= 0. Instead, block notation should be used to make it explicit:

Another block statement concern is that of where and how frequently to place explanatory comments in the code. Programmers who place comments in code in order to document it often wonder how much documentation is necessary. For example, should each line be commented? Probably not, because this most likely makes the source files difficult to read. In programming, as in life, less can be more.

Start by placing your comments before all control blocks. Typically, all the information you need to provide someone is best presented in a block-by-block fashion; therefore placing comments before blocks is a good rule of thumb to follow. Knowing what each block is supposed to do makes it easier for someone to check the code. More comments are not usually worth the added effort, unless a line does something particularly unusual or important.

Using `if-else` Statements to Make Decisions

The `if-else` branch is the basic control flow expression in the JavaScript language. Simply put, *if* something is true, you execute the first statement; *else*, do the other statement. For example:

```
if(a < b) a = b
else b = a
```

NOTE **Note that the expression evaluated is a *boolean*, not a number—unlike C, in which 0 = false and any other number is true. You must make some relational statement in order for JavaScript to evaluate the `if` statement. Because JavaScript automatically changes types to match the variable target, however, the zero and nonzero numbers are changed to false and true, respectively. This is *not* the case in Java, so keep that in mind if you plan to start learning and programming Java.**

```
statement1 {
    statement2 {
        statement3
        statement4
    }
    statement5 {
        statement6
        statement7
    }
}
```

The preceding code is much easier to keep track of than the following, in which there is no indentation:

```
statement1 {
statement2 {
statement3
statement4
}
statement5 {
statement6
statement7
}
}
```

In this last case, you would be searching for braces, trying to match them up with each other, and wondering why this code's author was so deliberately cruel.

NOTE There are many good editors for HTML out on the Web that support all the tags available and help keep track of additional media such as in-line pictures and sounds. A special text editor for writing your JavaScript code is useful, however—one that does automatic indentation, and also picks out matching braces so you can find the beginning and end of code blocks, as well as make sense out of expressions that use several layers of parentheses. A good program for Macintosh is a freeware text editor known as BBEdit Lite, which can be upgraded to the fully supported BBEdit version. BBEdit Lite is available at many shareware sites and directly from Bare Bones Software at http://www.barebones.com.

Instead of printing "second line of for loop i" after the first line in the loop, this line only appears once at the end of the output text. This problem occurs because the interpreter is not able to tell how many more statements it should run in the loop beyond the first one, so it just runs one. Otherwise, where would it stop? Two lines? Four? Ten? The way to guarantee that the interpreter knows how many lines to execute with the for statement is to use the braces.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript" >
<!-- hide script from old browsers
for(i=0;i<10;i++) {
    document.write("for loop " + i + "<br>")
    document.write("second line of for loop " + i + "<br>")
}
// end script hiding -->
</SCRIPT>
</BODY>
</HTML>
```

You will run across blocks of code similar to the preceding one in all sorts of statements, especially in organizing objects and functions, along with control flow statements. If you imagine them as setting off a series of statements that act as a single unit, you should be able to understand the program structure.

Using Indentation for Clarity

One thing you might have noticed is the use of indentation in the code examples. Indentation enables the reader to view statements in a block as related without searching for beginning and end braces. Because spaces and tabs are ignored by the interpreter anyway, they are simply there for the program reader's convenience. Indentation is, however, one of the most common elements of style used in programming, and you should definitely use it.

The standard for indentation and block usage is to indent the code within a block, but to leave the braces at the indentation level of the previous code, as follows:

It is not important to understand the exact nature of the function statement right now. It is important, however, to look at the use of the braces to tell where the function's statements begin and end. The use of braces enables the interpreter to organize and execute the different lines, and to associate them with the proper control flow statements and function declarations. You will also run into these code blocks when you have more than one statement after a control statement. The following example shows how a single statement works with the `for` loop. Don't worry yet about the exact syntax of the `for` loop; it is enough to understand that the command tells the interpreter to repeat the statements that follow 10 times.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript" >
<!-- hide script from old browsers
for(i=0;i<10;i++)
      document.write("for loop " + i + "<br>")
// end script hiding -->
</SCRIPT>
</BODY>
</HTML>
```

Run the preceding code and view the output. You should see a column of statements saying `for loop 0`, `for loop 1`,...`for loop 9`. Now, what if you wanted to add a statement to be executed in the loop? Your first impression might be to add the new line that you wanted right inside the loop. Try it:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript" >
<!-- hide script from old browsers
for(i=0;i<10;i++)
    document.write("for loop " + i + "<br>")
    document.write("second line of for loop " + i + "<br>")
// end script hiding -->
</SCRIPT>
</BODY>
</HTML>
```

Table 4.1 *Control Flow Expressions*

Type	Expression

`if-else`	`if(boolean)statement`
	`else statement`

`break`	`break [label]`
`continue`	`continue [label]`
`return`	`return expression`

`for`	`for(expression1 ; expression2 ; expression3)`
	`statement`

`for-in`	`for(expression1 in expression2)`
	`statement`

`while`	`while(boolean) statement`

In the next few sections, the `if-else`, `break`, `continue`, `for`, and `while` control flow structures are discussed. The `return` and `for-in` control flow statements, which are particular to objects and functions, are covered in their respective sections in the last half of this chapter, and in the next chapter. For now, a short statement about blocks and comments will help make your reading and writing of code much easier. Incorporating comments into your code early helps you get used to the idea of documenting your work and gets you accustomed to reading comments along with the code fragments to which they belong.

Using Blocks and Comments to Organize Code

The basic organizational method for grouping code uses the curly braces to define the beginning (`{`) and end (`}`) of code blocks. You might have noticed these braces in the function definitions from the previous chapter's examples. The following is an excerpt of that code:

```
Function functionName( arguments ) {
...
}
```

- □ *Methods.* A JavaScript object method is essentially a JavaScript function that is associated with a particular object. These methods are the ways in which the objects are manipulated and caused to carry out desired behaviors.

To use objects, you must create instances of the objects. Except for static objects, which are discussed later, the definition of an object is just that—a definition. The definition is a mold for the final object. To actually use an object, you create an object from a definition you have already written for it; this definition is similar to a JavaScript function. To create a copy of an object that has already been defined, you use the `new` operator:

```
birthday = new Date("August 24, 1972")
```

Notice that the `Date()` function takes a single argument—the date itself. The function that has the same name as the object is considered the ***constructor*** (described later in this chapter), and its arguments generally set the properties of the object. This concept goes for all of the objects that are built into the JavaScript language and the Netscape environment. After you create a new copy of the object, you can then use the methods and properties that are defined for it by using the name you have created for it. For example, anywhere you want to use the Date object `birthday`, you would simply use the new name.

```
yearBorn = birthday.getYear()
```

The advantage of this model is that you can create several instances of an object, and, although they each have different names and different properties, each has the same methods and behaves identically.

```
Christmas = new Date("December 25, 1995")
idesOfMarch = new Date("March 15, 1995")
month1 = Christmas.getMonth()
month1 = idesOfMarch.getMonth()
```

NOTE If you test these data functions, keep in mind that the getMonth() function returns a calendar number using a zero-based index. This means that January starts at 0 and December ends at 11. Blame computer science as a field if you want to hold someone accountable for this kind of counter-intuitive thought.

After you have created these objects, you'll want to gain access to their properties and methods.

Object Properties

Object properties are the variables in which objects hold information that either is being held for use in the program or is necessary for the execution of that particular object's methods. One of the most common uses for objects is the storage of often-used data and the particular methods that act on this data. Let's say you have an object that holds URLs and creates a hierarchical arrangement based upon information you provide to create a graphical interface for browsing a Web site. To create the hierarchical list, however, you must first be able to store information about each URL. For example, if the URL object includes information such as name, size, type, and modification date, in addition to the URL itself, the object, then, consists of the following properties:

Object:	`URLEntry`
Properties:	`name`
	`size`
	`type`
	`modDate`
	`URL`

Of course, the first method used to assign values to object properties is typically in the constructor itself, as shown in the following line:

```
homePage = new URLEntry("Hayden Books",2234,"folder","January 16,
1996 12:00:00", "http://www.mcp.com/hayden")
```

This particular constructor (URLEntry(...)) takes all the function arguments and creates the object with all of the properties set to the given argument values. Later on, when the creation of objects is covered, how constructors take their arguments and make objects is covered in more detail.

Assuming that the object already exists and is named homePage, you can access each of its properties in three ways.

First, you can use the dot operator (.) to access the properties, exactly as you would do in Java. The following code can be used to assign values to each of the member properties:

```
homePage.name = "Hayden Books"
homePage.size = 22345
homePage.type = "folder"
homePage.modDate = new Date("January 16, 1996 12:00:00")
homePage.URL = "http://www.mcp.com/hayden"
```

This format enables you to access each of the properties, set them, and use them later. This is the standard method for accessing properties in Java, and using it gets you used to the syntax of Java objects as well. However, there are two more ways of accessing JavaScript properties that may be more flexible in a dynamic environment. These methods involve accessing the member properties as array indices, and can be used when the exact name of the property needed is not known until runtime.

Second, you can access each property based upon its index:

```
homePage[0] = "Hayden Books"
homePage[1] = 22345
homePage[2] = "folder"
homePage[3] = new Date("January 16, 1996 12:00:00")
homePage[4] = "http://www.mcp.com/hayden"
```

Accessing the property based upon its index enables you to use control flow statements, such as for() loops, to cycle through all of the properties of an object. For example, you can use the following code to list all of the properties of an object:

```
function showProps(object) {
    for(var i = 0; i <5; i++)
```

```
        document.write(object[i])
}
```

In this case, however, you must know exactly how many property elements there are in an object to use such a function. Fortunately, JavaScript provides a control flow statement that cycles through all of the properties in an object by name. It is the `for-in` statement. We can redo the previous example using this method:

```
function showProps(object, name) {
    for(var prop in object)
        document.write(name+"."+prop+" = "+object[prop]+"\n")
}
```

Currently no JavaScript function exists to retrieve the name of an object, so we must pass an object's name property to any function or method that will refer to a specific object. In the case of our `homePage` object, we would call the `showProps` function with the statement:

```
showProps (homePage, "homePage")
```

The output from this call to `showProps` is

```
homePage.name = Hayden Books
homePage.size = 22345
homePage.type = folder
homePage.modDate = January 16, 1996 12:00:00
homePage.URL = http://www.mcp.com/hayden
```

The advantage of using the `for-in` control flow statement is that you do not need to know the number and names of the methods beforehand.

For the third method of accessing properties, notice the reference to the object property in the `document.write` statement. The array index uses the `prop` variable, which when used by itself is a string providing the property name for output. In the same way that the variable `prop` is a string, you can use strings to access and assign object properties.

```
homePage["name"] = "Hayden Books"
homePage["size"] = 22345
hoemPage["type"] = "folder"
homePage["modDate"] = new Date("January 16, 1996 12:00:00")
homePage["URL"] = "http://www.mcp.com/hayden"
```

This way, you can access properties at runtime without knowing their names. This enables you to address object properties dynamically in a script without knowing what the property names might be. You could, for example, use information from a text field to determine the properties a user wants to see.

Object Methods

In addition to being able to store properties, objects can also have *methods* associated with them. These methods are essentially JavaScript functions that are associated with an object and are used to carry out behaviors that the object encapsulates. To use a method that an object has built in, you must call it using the method reference. For example, to call the `sin()` method of the `Math` object which is built into JavaScript, you would call it:

```
Math.sin(3.14)
```

The method is `sin()`, which requires a number and is a member of the object `Math`. Many of the JavaScript and Netscape objects have methods associated with them, and the next three chapters are essentially presentations of these different objects, including the properties and the methods that they contain.

From the example of the `URLEntry` (discussed in the previous section about object properties), the `showProps` function shown here:

```
function showProps(object, name) {
    for(var prop in object)
        document.write(name+"."+prop+" = "+object[prop]+"\n")
}
```

... might be a method of `URLEntry`, renamed to `toString`. The only difference would be that instead of printing out the object property itself, the method might return a string that can then be used by the programmer to be printed to the screen or whatever else was needed.

```
    for(var prop in this)
        document.write(prop+" = "+object[prop]+"\n")
}
```

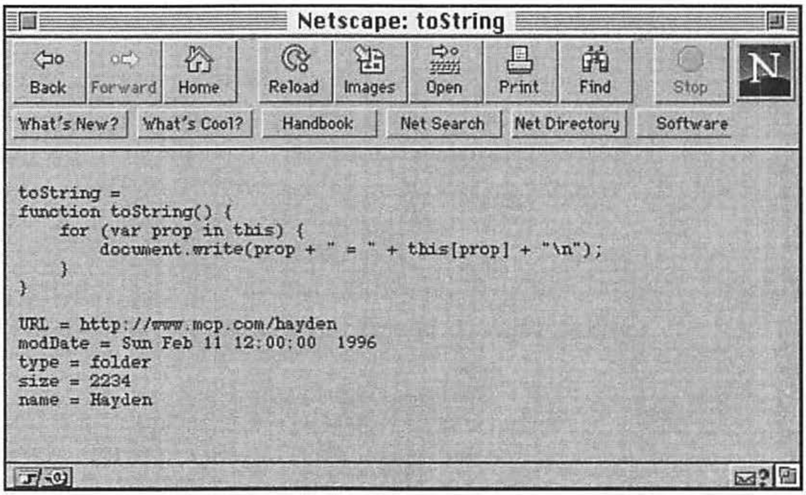
If the object name is newURL, you can get it to output the text by using the following call:

```
document.write(newURL.toString())
```

Some sample output from such a call is shown in Figure 5.1. Notice that the function definition for the toString method is also listed. Moreover, if you used the call:

```
document.write(newURL)
```

... you would get the exact same output. A call to document.write using any object invokes its toString function if it exists. This means that if any object you are using has a toString() method, you can send the object itself to document.write() for output. If it is an object of your own creation, you need to create your own toString() method, which is covered in the next section on creating objects.



```
toString =
function toString() {
    for (var prop in this) {
        document.write(prop + " = " + this[prop] + "\n");
    }
}

URL = http://www.mcp.com/hayden
modDate = Sun Feb 11 12:00:00  1996
type = folder
size = 2234
name = Hayden
```

Figure 5.1 *The* toString *method for the* URLEntry *object.*

Accessing Properties of Static Objects

There are several objects for which you do not need to create an instance of the object to use the methods and properties they contain. Such objects are considered *static*, and you can access their properties by using the object name itself and the property or method required. A perfect example of a static object is `Math`. When using the `Math` object, you do not need to create a new `Math` object with a new name. Instead, you can simply use the `Math` name, appended with the property or method you need.

```
area = Math.PI*Math.pow(r,2)
```

NOTE **Throughout this book, every attempt has been made to indicate whether an object is a static object or whether an instance of the object needs to be created.**

Using the `with` Statement to Refer to Objects

In addition to the standard use of methods, there is an important way to tell JavaScript which object you are referring to when calling methods and accessing properties. The statement to use is

```
with object {
...
}
```

All of the statements inside the with brackets act as if *`object`* was appended to the front. For example, you could use several method properties and methods without explicitly saying `Math.`*`member`* each time.

```
with Math {
    positive = (b+sqrt(4*a*c+pow(b,2)))/(2*a)
    negative = (b-sqrt(4*a*c+pow(b,2)))/(2*a)
}
```

If you did not use the `with` statement, you would be required to place `Math` in front of all of the method calls such as `sqrt()` and `pow()`.

Using this

In JavaScript and Java there is a keyword that refers to the current object—this. The current object is the object that contains the function, or in the case of a form, it is the form element object. An example of using this in JavaScript can be observed in the toString method for the URLEntry object.

```
function toString() {
    for(var prop in this)
        document.write(prop+" = "+object[prop]+"\n")
}
```

Notice in the for·in statement that the object whose properties are being traversed is this. In the case of the toString method, the for loop needs to know which object's properties to traverse; therefore, the this variable is used. You will see the use of this in many places as you move further into the use of objects.

Creating Objects

Although the use of objects that exist in the language and browser environment is a very powerful, time-saving practice, much of JavaScript's power comes from its capability to create new objects that carry out the jobs required for your Web pages. JavaScript's object model is somewhat different from Java's, as mentioned before; therefore, creating objects in JavaScript uses a much different methodology. Don't let this bother you if you don't plan on learning Java soon; the knowledge of objects you gain in this book is perfectly suitable to all your JavaScript needs. If you do choose to learn Java, you'll still have an important background in basic OOP concepts already built.

The JavaScript object is based around a constructor function that holds all of the properties and links to method functions. The JavaScript object does not provide any means for encapsulation in relation to property or method hiding, nor does it support abstraction or inheritance as in Java. While this is a limiting factor in terms of an object-oriented paradigm, the purpose of JavaScript is to simplify the Java view of objects. If the JavaScript developers had attempted to build in all of that functionality, then you'd basically be stuck with learning, well, all of Java.

The Constructor Function

To create a JavaScript object, you create a function that has the same name as the object you want to create. In the case of our `URLEntry` object, the constructor function is

```
function URLEntry(name, size, type, modDate, URL) {
    ...
}
```

This function definition can then be used to create instances of the object:

```
homePage = new URLEntry("Hayden Books",512,"folder","January 16, 1996
➥12:00:00", "http://www.mcp.com/hayden")
homePage = new URLEntry("Netscape",512,"folder","January 16, 1996
➥12:00:00", "http://home.netscape.com")
```

Adding Properties to Objects

Of course, to do anything with the arguments that the constructor function takes, you must create properties that can store data for the object. To add the variables that the arguments of the constructor function provides, you must assign the variables to member properties.

```
function URLEntry(name, size, type, modDate, URL) {
    this.name = name

    this.size = size

    this.type = type

    this.modDate = new Date(modDate)

    this.URL = URL

}
```

Notice the use of the Date object to hold the modification date of the URLEntry object. In addition to the standard variables, you can also have other objects as properties. The this variable name is used to indicate that the object to which the property being assigned is the object to which the constructor function belongs. Now, when the new URLEntry() constructor is called, the arguments passed to it by the program will be assigned to the properties of the object.

Adding Methods to Objects

In addition to adding properties to objects in the constructor, you need to add methods for the object so that it can actually do something. This is also carried out in the constructor function for an object. The URLEntry object, for instance, needs to be able to return a string representing the properties of the object. The function we need to add, therefore, is

```
function toString() {
    for(var prop in this)
        document.write(prop+" = "+object[prop]+"\n")
}
```

To add this function as a method of URLEntry, you can employ a similar statement as that used for properties.

```
this.toString = toString
```

Now, you can use the toString() function as a method of the URLEntry object. When you call an object of URLEntry type with the toString() method, it returns a string that holds all of the properties of the object instance.

There is now enough code to present an object that holds information and provides a string representation of itself when needed.

```
<HTML>
<HEAD>
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!-- begin hiding
function URLEntry(name, size, type, modDate, URL) {
    this.name = name
    this.size = size
    this.type = type
    this.modDate = new Date(modDate)
    this.URL = URL
    this.toString = toString
}

function toString() {
    for(var prop in this)
        document.write(prop + " = " + this[prop] + "\n")
}

newURL = new URLEntry("Hayden",2234,"folder","February 10, 1996
➥12:00:00", "http://www.mcp.com/hayden")
document.write(newURL)

// end hiding -->
</SCRIPT>
</PRE>
</HEAD>
<BODY>
</BODY>
</HTML>
```

NOTE Sometimes in lines of code you find that each of the statements ends with a semicolon. The semicolon is not a requirement of the JavaScript language, but *is* a requirement of Java. It doesn't matter whether you leave them off in JavaScript; however, if you plan to learn Java, realize that these are required and make a habit of including them in your code.

Creating Arrays

Most programming languages include a variable type known as an array. *Arrays* are collections of data under a single variable that are accessed using some kind of index. An array in most languages is a special data type; when you declare an array, you typically give it a variable name and specify how many indexes, or slots, the array will hold. For example, the following code would create an array with ten index slots in Microsoft Visual Basic:

```
dim basicArray(10) as Integer
```

This statement creates an array variable that has 10 slots, indexed from 0 to 9. Each slot can then hold a value that is an integer; notice that this language is more strongly typed than JavaScript and requires a type declaration for its variables, including arrays.

Although JavaScript has no explicit array data type as is found in Visual Basic or Java, you can create an array structure by defining an object `makeArray`. By doing so, you can create properties for your object to hold the data you need. The function declaration is

```
function makeArray(n) {
    this.length = n
    for(var i = 1; i <= n; i++)
        this[i] = 0

   return this
}
```

... `makeArray` actually creates a JavaScript object that acts, for all intents and purposes, like an array variable. In this case, the `makeArray` object has a `length` property which tells JavaScript how many slots it has. You call the `makeArray` function with the number of elements you want the array to hold. You can then assign values and access the individual slots of the `makeArray` object using the *`identifier`*`[`*`n`*`]` statement. The following code creates an array of three items and then prints them out on the display:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
function makeArray(n) {
    this.length = n;
    for(var i = 1; i <= n; i++)
        this[i] = 0

   return this
}

testArray = new makeArray(3)

testArray[1] = "hello\n"
testArray[2] = "this\n"
testArray[3] = "is\n"
```

```
document.write("<PRE>")

for(i=1;i<=3;i++)
    document.write(testArray[i])
document.write("length = " + testArray.length)
document.write("</PRE>")
// -->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

In addition to the standard variable types, you can also hold objects in arrays. As with all JavaScript statements, there is no need to explicitly declare the type of data you want the variable to hold, so the array can store all types.

One problem with the declaration of arrays as defined in the JavaScript beta documentation is that it sets a standard whereby arrays are indexed from 1 to n, where n is the number of elements in the array. Unfortunately, Java, C/C++, and most languages start indexing arrays at 0 and run to n–1. Because JavaScript objects store *all* properties in the array structure so that they can be indexed (as mentioned in the previous section on objects), the property length takes up this first (0) position; therefore, the data in the array must be placed in 1 to n.

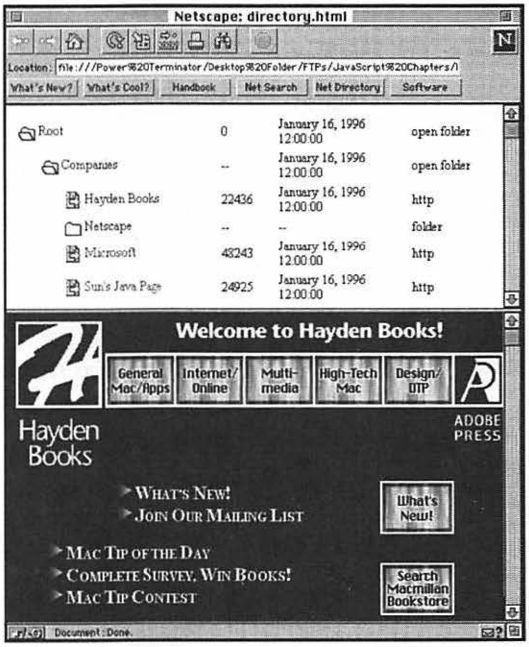
You can, in fact, reverse the order in the makeArray function between the for statement and the creation of the length property to make sure that the data indices fall from 0 to n–1, so that it conforms with the other language standards. You do this by moving the length property from the beginning of the constructor function to the end. This is how it looks:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
function makeArray(n) {
    for(var i = 1; i <= n; i++)
        this[i] = 0

    this.length = n
    return this
}

testArray = new makeArray(3)

testArray[0] = "hello"
testArray[1] = "this"
testArray[2] = "is"
for(i=0;i<3;i++)
    document.write(testArray[i])

document.write("length = " + testArray.length)

// -->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

NOTE In all of the examples, this book uses a method of indexing arrays that runs from 1 to n. Although you might have to relearn 0 to n-1 indexing for Java and for all of the arrays provided with built-in objects, using 1 to n in JavaScript reduces the opportunity for errors to appear in your own JavaScript code that creates arrays. (Just who was it that first taught computer scientists to count up from 0 anyway?)

Creating a Directory for a Web Page

There is now enough material to show how to bring all of the object elements together into a full-blown JavaScript program. In this example, the purpose of the script is to provide a Finder-like directory structure that can be used to organize a home page or even an entire Web site. The example shows some of the potential available with JavaScript, and ways in which it can be used to automate and enhance Web pages. Figure 5.2 shows what the final script provides.



Location: file:///Power%20Terminator/Desktop%20Folder/FTPs/JavaScript%20Chapters/

Root	0	January 16, 1996 12:00:00	open folder
Companies	--	January 16, 1996 12:00:00	open folder
Hayden Books	22436	January 16, 1996 12:00:00	http
Netscape	--	--	folder
Microsoft	48243	January 16, 1996 12:00:00	http
Sun's Java Page	24925	January 16, 1996 12:00:00	http

Figure 5.2 *The directory JavaScript program.*

To make this JavaScript work, you need to access three HTML files that are on your CD-ROM:

- □ directory.html. This file contains the all of the JavaScript code outlined below.

- □ clear document.html. This blank file is in the same folder as the directory.html file. It contains the following code:

```
<HTML>
<HEAD>
</HEAD>
</BODY>
</HTML>
```

- □ clear directory.html. This file is in the folder as directory.html and contains the following HTML code:

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<SCRIPT LANGUAGE="JavaScript">
```

```
<!--
parent.displayRoot()
// -->
</SCRIPT>
</BODY>
</HTML>
```

In addition, a number of .GIF image files are located in this same folder and are used by the JavaScript to create the icons you see in Figure 5.2.

The first function in the code is the makeArray() function, which is used to create arrays in JavaScript, as described in the previous section.

```
/**
 * this is used to create arrays in JavaScript
 */
function makeArray(n) {
    this.length = n

    for(var i = 1; i <= n; i++)
        this[i] = 0

    return this
}
```

As you can see, the 1 to n indexing is used because it presents a more robust object to deal with, even though it takes some getting used to if you are an experienced programmer.

The next function is the constructor function for the URLEntry object. This object is the basic element in the directory structure. Besides a few additional helper functions, all of the necessary behavior is coded into this object.

```
/**
 * this is the constructor function for the URLEntry object
 */
function URLEntry(name, size, type, modDate, URL, objectName) {
    this.name = name

    this.size = size

    this.type = type
    this.modDate = modDate

    this.URL = URL

    this.objectName = objectName

    this.URLEntry = new makeArray(0)

    this.display = displayURLEntry

    this.addURLEntry = addURLEntry

    this.toggle = toggle

}
```

When an object is created, it takes several parameters. The following list indicates the uses for these parameters.

- □ name. This is the name displayed for the URLEntry in the directory.
- □ size. This is the size of the file the link points to. This parameter can be used to indicate to users how long it might take to download the file. Of course, with some kinds of URL entries, like FTP and Mailto, there is no file size.
- □ type. This parameter indicates the type of entry, such as whether it is an HTML document, FTP site, folder, and so on. This string is used to indicate which GIF file should be loaded as the icon for the entry. The property also is displayed as text at the end of the entry. This value is used to hold the state of folder URLEntries. Also, if the type is "folder," then it can be toggled to "open folder" with the onClick event handler.
- □ modDate. This is the last modification date of the link, and is displayed on the URLEntry line in the directory.
- □ URL. This is the actual URL that the link points to, and is used as the target of an HREF tag when the URLEntry is selected.
- □ objectName. This is used to hold the actual name of the object when it is created. JavaScript doesn't have a built-in method for exposing an object's name, so in order to refer to it by

name, you must direct it to store its name, and then you can reference it. It is necessary to have an object's name in order to call the object's method from an event handler, such as onClick or onMouseOver.

- □ URLEntry. This is an array, initially set for zero elements, that will hold the children URLEntries of this URLEntry. When it comes time for a URLEntry to display itself, after doing so, it will instruct its children to display themselves. This displaying of children occurs only with folder URLEntries that are currently set as open folders.

In addition to the standard properties, the URLEntry object also has three methods that are assigned to it.

- □ The first method is display(), which in the case of the previous constructor function code is assigned to the function displayURLEntry(). This method tells the object to display itself.
- □ The second method, addURLEntry(), adds children to folder URLEntries. Of course, you can always add children to nonfolders, but they will never be displayed.
- □ The final method of the URLEntry object is toggle(), and is the event handler method called when an item is clicked. If the URLEntry object is a folder, toggle() either opens or closes the folder based upon its current state.

Each of these methods will now be covered in more detail. The following function displays the URLEntry object in the directory structure window.

```
/**
 * this is the display function for the URLEntry object
 */
function displayURLEntry() {

    frames[0].document.write("<TR>")

    frames[0].document.write("<TD WIDTH=200>")

    for(var i = 0; i<this.level; i++)
        frames[0].document.write("<IMG ALIGN=LEFT
SRC=\"blank.gif\">")

    frames[0].document.write("<A HREF=\"" + this.URL + "\" ")

    if(this.type == "folder" || this.type == "open folder")
        frames[0].document.write("TARGET=\"DIRECTORY_WINDOW\"")

    else
        frames[0].document.write("TARGET=\"DOCUMENT_WINDOW\"")

   frames[0].document.write("onClick=\"parent."+this.objectName+".toggle()\">")

    frames[0].document.write("<IMG ALIGN=LEFT BORDER=0 SRC=\"" +
    ➥this.type + ".gif\">")

    frames[0].document.write(this.name + "</A>")

    frames[0].document.write("</TD><TD WIDTH = 50>")

    frames[0].document.write(this.size)

    frames[0].document.write("</TD><TD WIDTH = 200>")

    frames[0].document.write(this.modDate)

    frames[0].document.write("</TD><TD WIDTH = 100>")

    frames[0].document.write(this.type)

    frames[0].document.write("</TD></TR>")

    if(this.type == "open folder") {
        for(var i = 1; i <= this.URLEntry.length; i++)
            this.URLEntry[i].display()

    }
}
```

The first thing you should notice is all of the `frames[0]` objects to which the `document.write()` method refers. This script is in the overall HTML file that creates the directory structure frame and display frame, and later in the code you will see where these frames are created. Because there are now essentially two subdocuments in the main window, it is necessary to indicate which document receives the text. The text sent to the document is HTML code for the creation of a row in a table. Later in this chapter, you will see the code that creates the beginning and end tags for the table.

NOTE Frames are a new specification for Netscape 2.0 and are not included in the HTML 3.0 standard. Check out Netscape's Web site for more information on how to implement frames in a document: **http://www.netscape.com/navigate/understanding_frames.html.**

The first column in the table row is set at 200 pixels wide, and is used to hold the icon and name of the `URLEntry` object. Before the icon and entry name are displayed, it is necessary to indent them depending upon the level of the hierarchy in which they occur. The first `for` loop creates a blank.gif image tag for every level of index. Next, if the `URLEntry` object is a folder, then the URL to which it points needs to update the frame holding the directory structure and not the document window. If the `URLEntry` object points to anything else, then the target of the URL to which it points will need to be updated in the lower document window.

NOTE The TARGET HTML argument is a new feature of Netscape 2.0. For more information on how to implement targeting in an HTML document, refer to Netscape's Web site: **http://www.netscape.com/eng/mozilla/2.0/relnotes/demo/target.html.**

The next lines create the HTML HREF tag so that you can create a link out of the `URLEntry` name. The tag is then used to update the directory structure whenever any of the `URLEntry` objects receive a mouse click. Notice the use of the double quote backslash control character (`\"`) used to send a double quote to the HTML document page.

In addition to the normal HTML tag accoutrements, the HREF tag has an extra argument: `onClick`. This argument tells what JavaScript statements are executed when the `Click` event occurs for this link object. The `URLEntry` object passes its own name to the HTML document, appending the method `toggle()` to the end—a procedure that causes the event handler to call the toggle statement from the correct instance of the object. This process is an important element of the program and one you should understand and experiment with.

It is important that the objects you want to call in a statement such as onClick are available to the event handler from the point of view of the document. If the onClick method had been this.toggle(), how does the event handler know what this is, and how do you put this into a string for an HTML document? By allowing the object to have a copy of its own name, it can pass the name on to the HTML document easily.

Following the construction of the HREF link, which controls the behavior of the directory structure, the next several lines create the rest of the table row. This row includes information about size, type, and modification date. The last three lines of the display function check to see whether the URLEntry is an open folder. If it is, then all (if any) of its children are told to display themselves.

```
/**
 * this function is used to add an entry to the folder type of
➥URLEntry
 */
function addURLEntry(newURLEntry) {
    newURLEntry.level = this.level + 1

    var tempArray = this.URLEntry

    this.URLEntry = new makeArray(this.URLEntry.length+1)

    for(var i = 1; i <= this.URLEntry.length; i++)
        this.URLEntry[i] = tempArray[i]

    this.URLEntry[this.URLEntry.length] = newURLEntry

}
```

The addURLEntry function is used to add entries to the URLEntry object and is specifically meant to be used with a folder object type. The function takes a URLEntry object as its argument. First, it sets the level to one higher than that of the previous URLEntry object level. If, for example, the folder to which you were adding was on level 2 of the directory hierarchy, then anything placed in it would be on level 3. This technique was used in the previous display method to provide indentation as a visual clue. Second, the next four lines take the original URLEntry array in the object, place it in a temporary array, enlarge the original array by one, and then move each entry back into place. Finally, the last line adds the new URLEntry to the newly created end of the list.

The following method is the method called by the onClick event handler in order to change the state of a folder.

```
/**
 * this function toggles the folder from open to closed and vice
➥versa
 */
function toggle() {
    if(this.type == "folder")
        this.type = "open folder"

    else if(this.type == "open folder")
        this.type = "folder"

}
```

The function simply checks the state of the URLEntry object. If the object is in a folder state (that is, closed), then it sets it to open. If it is open, then it resets it to a closed state. If the object is not a folder, it simple falls through the method. After the toggle method is called, the link for the HREF tag is called. If the object is a folder, then the target becomes the directory window itself. The link is a document called clear directory.html. This file is an HTML file with only the necessary headers and a single function, which is to displayRoot(). This essentially blanks the window and calls the displayRoot() function to redraw the screen, allowing the directory with the updated folder structure to be seen. The clear directory.html file is as follows:

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<SCRIPT LANGUAGE="JavaScript">
<!--
parent.displayRoot()

// -->
</SCRIPT>
</BODY>
</HTML>
```

Notice that the displayRoot() method call is directed at the parent object. In this case, the parent is the entire window that holds the two frames. It is the parent of these frames that is holding all of the JavaScript code, and where objects in this case need to be referenced.

```
/**
 * This is the declaration for the root URLEntry which starts it all
 */
root = new URLEntry("Root", 0, "folder","January 16, 1996
➥12:00:00","clear directory.html","root")


/**
 * this is the function called to create the table, and direct each of
 * the URLEntries to display themselves and their children
 */
function displayRoot() {
    frames[0].document.write("<HTML><BODY BGCOLOR=\"#FFFFFF\">")

    frames[0].document.write("<TABLE BORDER=0>")

    root.display()

    frames[0].document.write("</TABLE>")

    frames[0].document.write("</BODY></HTML>")

}
```

The previous two sections of code create the root object that holds all of the URLEntry object (and is a URLEntry object itself) and begin displaying the entire tree. The construction of the root entry is like any other object constructor, and provides all of the arguments the entry needs to create the object. There is one piece of code that is essential, however, and that is the URL itself. It is important that this URL point to the clear directory.html file that is used to clear the screen and trigger a redraw.

The displayRoot() function is necessary in order to get the URLEntry hierarchy going. It is necessary to create the beginning and end tags for the table that the display() method provides the rows for. Note that the displayRoot function is not attached to any object, and is invoked as a method for setting up the table and triggering the display of the hierarchy—which, once started, traverses the tree of `URLEntries` and displays them as directed.

The following function, as noted in the reminder code, is used to create the two frames that hold the directory and document windows.

```
/**
 * this function creates the directory and document frames in which
 * everything is displayed
 */
function createFrames() {
    document.write("<FRAMESET ROWS=\"150,*\">")

    document.write("<FRAME SRC=\"clear directory.html\"
NAME=\"DIRECTORY_WINDOW\">")

    document.write("<FRAME SRC=\"clear document.html\"
NAME=\"DOCUMENT_WINDOW\">")

    document.write("</FRAMESET>")

}
```

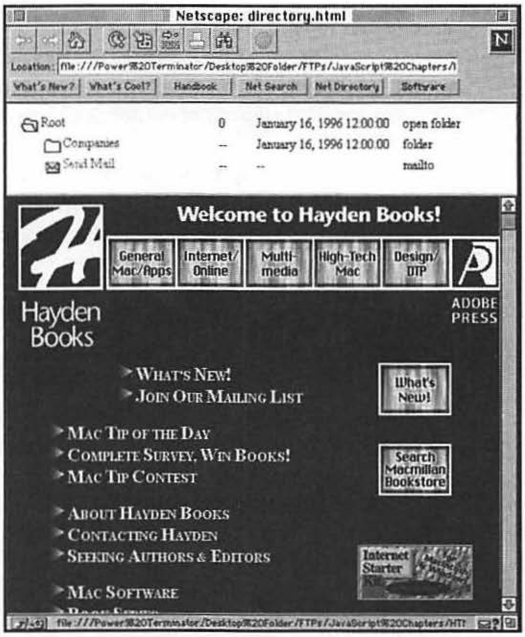
The frames are created and two source files are used in order to provide clear windows on loading. Of course, the clear directory file also calls the function `displayRoot()` that causes the window to be updated.

This is essentially all of the code you need to create and control the directory. Now, all that is necessary are some URLs to provide to the `URLEntry` object. The following code includes the constructors used in building up the directory shown in Figures 5.2 through 5.4.

```
/**
 * this is the beginning of the creation of the directory structure
 * note: all of these entries are dummies, and you can change them to
 * your own.
 */

//add companies Folder to root
compFolder = new URLEntry("Companies","--","folder","January 16, 1996
➥12:00:00","clear directory.html","compFolder")
```

```
root.addURLEntry(compFolder)


//add Hayden Books url to companies folder
haydenURL = new URLEntry("Hayden Books",22436,"http","January 16,
➥1996 12:00:00","http://www.mcp.com/hayden/","haydenURL")

compFolder.addURLEntry(haydenURL)

//add Netscape folder to companies
netscapeFolder = new URLEntry("Netscape","--","folder","--","clear
➥directory.html","netscapeFolder")
compFolder.addURLEntry(netscapeFolder)

//add Netscape WWW to Netscape folder
netscapeURL = new URLEntry("Netscape WWW",34067,"http","January 17,
➥1996 12:00:00", "http://home.netscape.com/","netscapeURL")
netscapeFolder.addURLEntry(netscapeURL)

//add Netscape ftp to Netscape folder
netscapeFtpURL = new URLEntry("Netscape FTP","--","ftp","January 17,
➥1996 12:00:00", "http://ftp.netscape.com/","netscapeFtpURL")
netscapeFolder.addURLEntry(netscapeFtpURL)

//add Netscape ftp2 to Netscape folder
netscapeFtp2URL = new URLEntry("Netscape FTP2","--","ftp","January
➥17, 1996 12:00:00", "http://ftp2.netscape.com/","netscapeFtp2URL")
netscapeFolder.addURLEntry(netscapeFtp2URL)

//add Netscape ftp3 to Netscape folder
netscapeFtp3URL = new URLEntry("Netscape FTP3","--","ftp","January
➥17, 1996 12:00:00", "http://ftp3.netscape.com/","netscapeFtp3URL")
netscapeFolder.addURLEntry(netscapeFtp3URL)

//add Microsoft url to companies folder
microsoftURL = new URLEntry("Microsoft",48243,"http","January 16,
➥1996 12:00:00", "http://www.microsoft.com/","microsoftURL")
compFolder.addURLEntry(microsoftURL)

//add Sun url to companies folder
javaURL = new URLEntry("Sun's Java Page",24925,"http","January 16,
➥1996 12:00:00", "http://java.sun.com/","javaURL")
compFolder.addURLEntry(javaURL)

//add mailto: url to root folder
mailURL = new URLEntry("Send Mail", "--", "mailto","--
➥","mailto:mshobe@u.washington.edu","mailURL")
root.addURLEntry(mailURL);
```

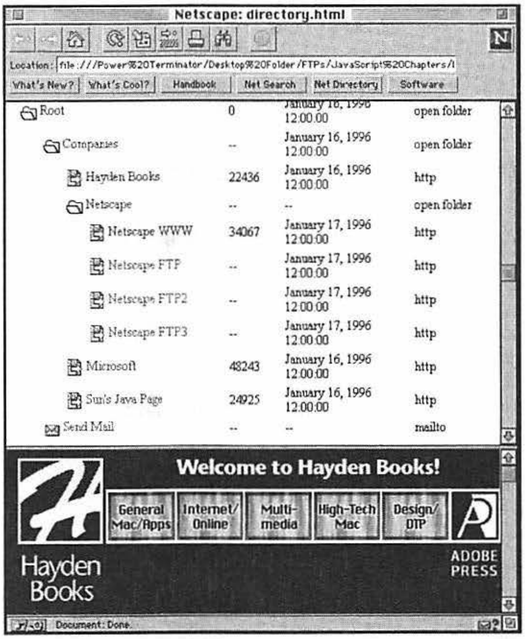
The final bit of code invokes the `createFrames()` function to start the whole thing in motion. When `createFrames()` is called, it creates the two frames in the window. The document frame sits idle, while the directory frame issues a call to the `displayRoot()` function. This function creates a table tag and calls on the `root` object to display itself, which in turn tells all of its children to display themselves, and so on, until all `URLEntries` that should be drawn are drawn. After the display is finished, the `displayRoot()` method ends the table tag, and the whole document is displayed. When one of the folder entries is hit, it triggers the `toggle()` method, which switches the state of the folder. At the same time, the URL of the pointer reloads the clear directory.html file that issues the call to `displayRoot()` again to update the display. If an entry besides a folder is selected then the URL is sent to the document window for display.



Location: file:///Power%20Terminator/Desktop%20Folder/FTPs/JavaScript%20Chapters/

Root	0	January 16, 1996 12:00:00	open folder
Companies	--	January 16, 1996 12:00:00	folder
Send Mail	--	--	mailto

Figure 5.3 *The directory script with a collapsed folder.*



Location: file:///Power%20Terminator/Desktop%20Folder/FTPs/JavaScript%20Chapters/I

Root	0	January 16, 1996 12:00:00	open folder
Companies	--	January 16, 1996 12:00:00	open folder
Hayden Books	22436	January 16, 1996 12:00:00	http
Netscape	--	--	open folder
Netscape WWW	34067	January 17, 1996 12:00:00	http
Netscape FTP	--	January 17, 1996 12:00:00	http
Netscape FTP2	--	January 17, 1996 12:00:00	http
Netscape FTP3	--	January 17, 1996 12:00:00	http
Microsoft	48243	January 16, 1996 12:00:00	http
Sun's Java Page	24925	January 16, 1996 12:00:00	http
Send Mail	--	--	mailto

Figure 5.4 *The directory script with an expanded folder.*

The entire file, when it is placed in the HTML tags, looks like the following:

```
<!--
    File: Directory.html
    Description: This script creates a directory structure for a Web
        page that can be automatically expanded and collapsed
    Author: Tim Ritchey
    Date: 16 January, 1996
    Note: This Script is freely available for download and use.
    Please indicate where you got it from, and I would be grateful.
-->
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!-- begin hiding

/**
 * this is used to create arrays in JavaScript
 */
function makeArray(n) {
    this.length = n
    for(var i = 1; i <= n; i++)
```

```
        this[i] = 0
    return this
}

/**
 * this is the constructor function for the URLEntry object
 */
function URLEntry(name, size, type, modDate, URL, objectName) {
    this.name = name
    this.size = size
    this.type = type
    this.modDate = modDate
    this.URL = URL
    this.objectName = objectName
    this.level = 0
    this.URLEntry = new makeArray(0)
    this.display = displayURLEntry
    this.addURLEntry = addURLEntry
    this.toggle = toggle
}

/**
 * this is the display function for the URLEntry object
 */
function displayURLEntry() {

    frames[0].document.write("<TR>")
    frames[0].document.write("<TD WIDTH=200>")
    for(var i = 0; i<this.level; i++)
        frames[0].document.write("<IMG ALIGN=LEFT
SRC=\"blank.gif\">")
    frames[0].document.write("<A HREF=\"" + this.URL + "\" ")
    if(this.type == "folder" || this.type == "open folder")
        frames[0].document.write("TARGET=\"DIRECTORY_WINDOW\"")
    else
        frames[0].document.write("TARGET=\"DOCUMENT_WINDOW\"")
   frames[0].document.write("onClick=\"parent."+this.objectName+".toggle()\">")
    frames[0].document.write("<IMG ALIGN=LEFT BORDER = 0 SRC=\"" +
    ➥this.type + ".gif\">")
    frames[0].document.write(this.name + "</A>")
    frames[0].document.write("</TD><TD WIDTH = 50>")
    frames[0].document.write(this.size)
    frames[0].document.write("</TD><TD WIDTH = 200>")
    frames[0].document.write(this.modDate)
    frames[0].document.write("</TD><TD WIDTH = 100>")
    frames[0].document.write(this.type)
    frames[0].document.write("</TD></TR>")
    if(this.type == "open folder") {
        for(var i = 1; i <= this.URLEntry.length; i++)
            this.URLEntry[i].display()
    }
}

/**
 * this function is used to add an entry to the folder type of
➥URLEntry
 */
function addURLEntry(newURLEntry) {
    newURLEntry.level = this.level + 1
    var tempArray = this.URLEntry
    this.URLEntry = new makeArray(this.URLEntry.length+1)
    for(var i = 1; i <= this.URLEntry.length; i++)
        this.URLEntry[i] = tempArray[i]
    this.URLEntry[this.URLEntry.length] = newURLEntry

}

/**
 * this function toggles the folder from open to closed and vice
➥versa
 */
function toggle() {
    if(this.type == "folder")
        this.type = "open folder"
    else if(this.type == "open folder")
        this.type = "folder"
}

/**
 * This is the declaration for the root URLEntry which starts it all
 */
root = new URLEntry("Root", 0, "folder","January 16, 1996
➥12:00:00","clear directory.html","root")

/**
 * this is the function called to create the table, and direct each of
 * the URLEntries to display themselves and their children
 */
function displayRoot() {
    frames[0].document.write("<HTML><BODY BGCOLOR=\"#FFFFFF\">")
    frames[0].document.write("<TABLE BORDER=0>")
    root.display()
    frames[0].document.write("</TABLE>")
    frames[0].document.write("</BODY></HTML>")
}
```

```
/**
 * this function creates the directory and document frames in which
 * everything is displayed
 */
function createFrames() {
    document.write("<FRAMESET ROWS=\"150,*\">")
    document.write("<FRAME SRC=\"clear directory.html\"
NAME=\"DIRECTORY_WINDOW\">")
    document.write("<FRAME SRC=\"clear document.html\"
NAME=\"DOCUMENT_WINDOW\">")
    document.write("</FRAMESET>")
}

/**
 * this is the beginning of the creation of the directory structure
 * note: all of these entries are dummies, and you can change them to
 * your own.
 */

//add companies Folder to root
compFolder = new URLEntry("Companies","--","folder","January 16, 1996
➥12:00:00","clear directory.html","compFolder")
root.addURLEntry(compFolder)

//add Hayden Books url to companies folder
haydenURL = new URLEntry("Hayden Books",22436,"http","January 16,
➥1996 12:00:00", "http://www.mcp.com/hayden/","haydenURL")
compFolder.addURLEntry(haydenURL)

//add Netscape folder to companies
netscapeFolder = new URLEntry("Netscape","--","folder","--","clear
➥directory.html","netscapeFolder")
compFolder.addURLEntry(netscapeFolder)

//add Netscape WWW to Netscape folder
netscapeURL = new URLEntry("Netscape WWW",34067,"http","January 17,
➥1996 12:00:00", "http://home.netscape.com/","netscapeURL")
netscapeFolder.addURLEntry(netscapeURL)

//add Netscape ftp to Netscape folder
netscapeFtpURL = new URLEntry("Netscape FTP","--","ftp","January 17,
➥1996 12:00:00", "http://ftp.netscape.com/","netscapeFtpURL")
netscapeFolder.addURLEntry(netscapeFtpURL)

//add Netscape ftp2 to Netscape folder
➥netscapeFtp2URL = new URLEntry("Netscape FTP2","--","ftp","January
➥17, 1996 12:00:00", "http://ftp2.netscape.com/","netscapeFtp2URL")
netscapeFolder.addURLEntry(netscapeFtp2URL)

//add Netscape ftp3 to Netscape folder
netscapeFtp3URL = new URLEntry("Netscape FTP3","--","ftp","January
➥17, 1996 12:00:00", "http://ftp3.netscape.com/","netscapeFtp3URL")
netscapeFolder.addURLEntry(netscapeFtp3URL)

//add Microsoft url to companies folder
microsoftURL = new URLEntry("Apple",48243,"http","January 16, 1996
➥12:00:00", "http://www.apple.com/","appleURL")
compFolder.addURLEntry(microsoftURL)

//add Sun url to companies folder
javaURL = new URLEntry("Sun's Java Page",24925,"http","January 16,
➥1996 12:00:00", "http://java.sun.com/","javaURL")
compFolder.addURLEntry(javaURL)

//add mailto: url to root folder
mailURL = new URLEntry("Send Mail", "--", "mailto","--
➥","mailto:tdr20@cus.cam.ac.uk","mailURL")
root.addURLEntry(mailURL)

createFrames()
// end hiding -->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Summary

This chapter covered the basics of creating your own objects in JavaScript and showed you how to use built-in objects as well. The sample script used to create the directory frame in a Web page is a fairly complex script that depends upon many features and objects built into the scripting language, and provides a good example of what can be done using JavaScript to enhance your own pages beyond the standard HTML standard. The next two chapters will cover the functions and objects that are built into the JavaScript language and the Netscape environment. Because JavaScript exposes many of the low-level features of displaying Web pages, you can wring the most out of plain HTML without requiring CGI scripts or plug-ins like Macromedia Shockwave to get great results.

CHAPTER 6

JavaScript and Built-In Objects

The previous chapters presented the groundwork for developing powerful Web-based documents that can dynamically adjust to environment and user events. Chapter 2, "Introduction to the Java Family," presented an introduction to the language features of Java and JavaScript and discussed the different ways in which each language provides support for client-side execution of code in Web documents. Chapter 3, "JavaScript Fundamentals," provided the fundamental look and feel of the JavaScript language and focused upon the overall architecture of writing JavaScript programs. Chapter 4, "Control Flow and Functions in JavaScript," presented more advanced features of the language, including control flow and the use of functions. Chapter 5, "Using and Creating Objects in JavaScript," discussed the most powerful feature of the JavaScript language—its object-based approach to script design and creation. The directory script shown at the end of that chapter is an excellent example of how an object such as `URLEntry` can provide functional interfaces for Web documents.

The real power in object-oriented computing is, of course, the capability to use already existing objects to extend the functionality of your programs without the need to code everything from scratch. Even with the basics of the JavaScript language behind us, it was necessary in previous chapters to use objects and methods we hadn't discussed before in great detail, because to do anything in a complex environment, such as the Netscape browser, it would be difficult to program every feature you needed from scratch. This chapter and the next two focus on those and other objects that the JavaScript language and Netscape browser provide. One of the best examples is the document object used to control the actual HTML content being displayed with its `write()` method.

The objects available for use in Web page scripts can be divided into two types:

- □ Objects built into JavaScript
- □ Objects provided by the Netscape environment

Objects that are built into the JavaScript language include `String`, `Math`, and `Date`. In addition, JavaScript provides three standalone functions: `eval`, `parseInt`, and `parseFloat`. This chapter focuses on these built-in objects and functions. Chapters 7 and 8 deal with objects provided by the Netscape environment that can be used in JavaScript programs. These objects include the window objects, such as `parent` and `frames`, and document and form objects, as well as the history and location objects.

Built-In Objects and Functions

JavaScript provides several objects that are considered a part of the standard language and are available in all run-time environments. These include the `String`, `Math`, and `Date` objects. In addition to full objects, JavaScript also provides several built-in, standalone functions such as `eval`, `parseInt`, and `parseFloat`. The `String`, `Math`, and `Date` objects are used in JavaScript to provide extended data type functionality for common formats of information not provided by the basic types such as characters, numbers, and booleans. In addition, the `Math` object provides methods for arithmetic functions not provided by the standard operators. The `eval`, `parseInt`, and `parseFloat` functions all return the numerical values for strings representing numerical values. These can be important when using values returned from form items, for example, text areas.

Unlike the Netscape Navigator objects that are discussed in Chapters 7 and 8, the objects provided by the JavaScript language are available automatically and can be expected to be available in every document. The Netscape Navigator objects are document-dependent and might change depending on the HTML environment.

Besides the standard objects related to HTML pages, a Netscape environment might run different plug-ins or applets, which provide even more objects with which JavaScript will in the future be able to interact. These plug-in- or applet-associated objects are currently unavailable in the JavaScript version embedded in Netscape 2.0, but they are promised to be available for future releases.

In addition, if support for JavaScript is provided by other Web browsers, it is possible that these browsers may expose objects for the Web document author to control using JavaScript. As object-oriented technology becomes more pervasive in application development and Web site creation, JavaScript's functionality can expand to incorporate these new objects into its realm of control. This means that your time investment in learning JavaScript's object-based environment can be applied to emerging Web technologies and will be of lasting value.

The `String` Object

The most commonly used object in JavaScript is the `String` object. This is because the `String` object does not need new statements to create an instance of the object. Any time you place text between two quotation marks and assign it to a variable or property, you create a `String` object. Each of the following lines, for example, create new `String` objects:

```
employeeName = "John Doe"
URLEntry.type = "folder"
```

Additionally, the use of a string literal creates an instance of the `String` object.

To use a string property or method, you append the dot and method or property name as in all objects.

```
stringLength = employeeName.length

tempString = URLEntry.type.toUpperCase()
```

One item to notice is that the dot property is cumulative and works from left to right, evaluating each statement and acting as if the next operation were appended to the result. The preceding statement, for example, also could be written:

```
tempString = URLEntry.type
temp2String = tempString.toUpperCase()
```

The evaluation works from left to right so that the method or property in question refers to a valid argument. In the preceding case, the `URLEntry` object property type is assigned to the variable `tempString`, which is then converted to all uppercase letters using the `toUpperCase()` method of the `String` object. This is essentially what happens with the statement that strings all the dot operations together.

You do not need to use a variable name for invoking methods or properties of strings. You simply can append the method or property name to the end of the string.

```
tempNumber = "Hello, World!".length
tempString = "Hello, World!".toUpperCase()
```

NOTE Of course, the previous code is an example to show you how literal strings can be used rather than to show you a procedure of practical importance. Normally, you would never want to perform a function on a string literal when you could figure out the result in advance and not avoid a function call at run time. Where possible, figure out what the result should be at design time and use that. Invoking unnecessary methods at run time is a waste of computing cycles, particularly when working in an interpreted environment like JavaScript where it is important to optimize scripts.

`String` Object Properties

The `String` object has a single property, `length`. This property holds the number of characters in the string, including all white spaces and special characters. The following code uses the `length` property of the `String` object to line up different columns of text:

```
<HTML>
<HEAD>
```

```
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!-- begin hiding
function spacer(string,n,type) {
    space = ""
    for(var i = 0; i < (n - string.length); i++)
        space += " "
    if(type == "L")
        return string + space
    else return space + string
}

document.write(spacer("Hayden Books", 25, "L"))
document.write(spacer("WWW Home Page", 15, "L"))
document.writeln(spacer("http://www.mcp.com/hayden", 30, "R"))

document.write(spacer("Netscape", 25, "L"))
document.write(spacer("FTP site", 15, "L"))
document.writeln(spacer("ftp://ftp.netscape.com", 30, "R"))


// end hiding -->
</SCRIPT>
</PRE>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Notice the use of the <PRE></PRE> tags to allow formatting of the text. Notice also the use of the `writeln()` method at the end of each row to include a carriage return (which may bring back unpleasant memories for former ANSI Pascal users). The `spacer` function uses the `length` property of the string to compute how many spaces to add to the beginning or end of the string to make sure they line up.

String Methods

In addition to the `length` property, the `String` object has several methods associated with it. We have seen a couple of the methods, such as `toUpperCase()` and `toLowerCase`, but what follows is an exhaustive list of all the `String` methods:

□ `anchor(`*`nameAttribute`*`)`. The `anchor` method creates an anchor out of the string using `nameAttribute` as the anchor NAME tag.

The following two lines of code produce the same HTML output; the first line uses standard HTML, while the second uses the anchor method:

```
document.write("<A NAME=\"anchor_point\">Hello, World!</A>")
document.write("Hello, World!".anchor("anchor_point"))
```

□ `big()`. The `big` method returns the string surrounded by the <BIG></BIG> tags that display the text in a bigger font relative to the base font size.

□ `blink()`. The `blink` method returns the string surrounded by the <BLINK></BLINK> tags. This causes the string text to blink in the resulting HTML document.

□ `bold()`. The `bold` method returns the string surrounded by the <B></B> tags, which causes the string to display in bold in the resulting HTML document.

□ `charAt(`*`index`*`)`. The `charAt` method returns the character in the string at the specified index.

□ `fixed()`. The `fixed` method returns the string surrounded by the <TT></TT> tags that display the string in a fixed-pitch font in the browser window.

□ `fontcolor(`*`color`*`)`. The `fontcolor` method surrounds the string with the color tag and causes the browser to display the string in the specified color. You can specify the red, green, and blue colors with the hexadecimal "rrggbb" color declaration or use one of the predefined colors. The resulting tag is <FONT COLOR="*color*"></FONT>.

□ `fontsize(`*`size`*`)`. The `fontsize` method surrounds the string with the <FONTSIZE=*size*></FONTSIZE> tag. You can use an absolute value between 1 and 7 or use a +/- in front of the number to indicate a relative font to the base font size declared in the BASEFONT tag. The default base font size is 3.

□ `indexOf(`*`character`*`, [`*`fromIndex`*`])`. The `indexOf` method searches the `String` object for the first occurrence of the character and returns that index. The `fromIndex` argument is optional and indicates where to begin the search. You could find the index values for all of the same characters, for example, by starting after the previous index.

```
while(index < text.lastIndexOf("a"))
    index = text.indexOf("a", index + 1)
    document.writeln(index + "\t")
```

- □ `italics()`. This method surrounds the string with the <I></I> tags that cause the text to display in italic font.

- □ `lastIndexOf(`*`character`*`, [`*`fromIndex`*`])`. The `lastIndexOf` method is identical to the `indexOf` method except that it searches through the string backward for the character starting at `fromIndex`.

- □ `link(`*`URL`*`)`. The `link` method is similar to the `anchor` method except it creates an HREF tag that points to the URL provided. The resulting tag is <A HREF=*URL>text*</A>.

- □ `small()`. This method is similar to the `big` method, except that it causes the string to be surrounded by the <SMALL> </SMALL> tags that display the text in a smaller font relative to the base font.

- □ `strike()`. The `strike` method causes the string to display with a strikeout through the middle and surrounds it with the <STRIKE></STRIKE> tags.

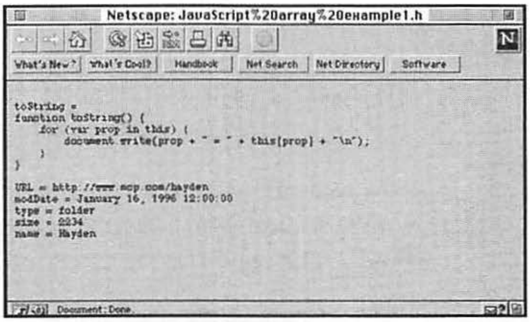
- □ `sub()`. The `sub` method causes the string to display in subscript by appending the <SUB></SUB> tags to the beginning and end of the string.

- □ `substring(`*`startIndex,endIndex`*`)`. The `substring` method returns the string that runs from `startIndex` to the character right before `endIndex`. If `startIndex` is larger than `endIndex`, the method acts as if the two are in each other's positions. If the two indexes are equal, it returns an empty string.

- □ `sup()`. The `sup` method returns the string with the <SUP> </SUP> tags appended to the beginning and end of the string. This causes the string to display in superscript.

- □ `toLowerCase()`. The `toLowerCase` method returns the string with all characters changed to lowercase.
- □ `toUpperCase()`. The `toUpperCase` method returns the string with all characters changed to uppercase.

Figure 6.1 shows what these methods each produce in the Hello, World! text.



```
toString =
function toString() {
    for (var prop in this) {
        document.write(prop + " = " + this[prop] + "\n");
    }
}

URL = http://www.mcp.com/hayden
modDate = January 16, 1996 12:00:00
type = folder
size = 2234
name = Hayden
```

Figure 6.1 *Results of the* `String` *methods on Hello, World!*

The script that follows produces the document.

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!--
text = "Hello, World!"

document.writeln(text.anchor("text"))
document.writeln(text.big())
document.writeln(text.blink())
document.writeln(text.bold())
document.writeln(text.charAt(5))
document.writeln(text.fixed())
document.writeln(text.fontcolor("red"))
document.writeln(text.fontsize(5))
document.writeln(text.fontsize(-2))
index = 0
while(index < text.lastIndexOf("l"))  {
    index = text.indexOf("l",index + 1)
    document.write(index +"\t")
}
```

```
document.write("\n")
document.writeln(text.italics())
document.writeln(text.link("http://www.mcp.com/hayden"))
document.writeln(text.small())
document.writeln(text.strike())
document.writeln("SUB" + text.sub())
document.writeln(text.substring(7,12))
document.writeln("SUP" + text.sup())
document.writeln(text.toLowerCase())
document.writeln(text.toUpperCase())

// -->
</SCRIPT>
</PRE>
</BODY>
</HTML>
```

The Math Object

The Math object provides standard mathematical constants and functions beyond what is provided by the standard operators for addition, subtraction, division, multiplication, and so on, where a single character represents the function to be carried out. The Math object is considered *static*; that is, there does not need to be an instance of the object in order to use the Math object's properties and methods. There is no need to use the new statement to create a copy of the Math object; instead, refer to the Math object directly.

```
area = Math.PI*Math.pow(radius,2)
```

NOTE The Math object is a perfect example of when to use the with() { ... } statement. If you are using several Math functions and constants together, it would be more convenient to declare a block as being with(Math) so that you can refer to the properties and methods of Math without prefixing the Math object name to every call. The following two code blocks are equivalent to each other.

```
With(Math){
area = PI*pow(radius,2)
}

area = Math.PI*Math.pow(radius,2)
```

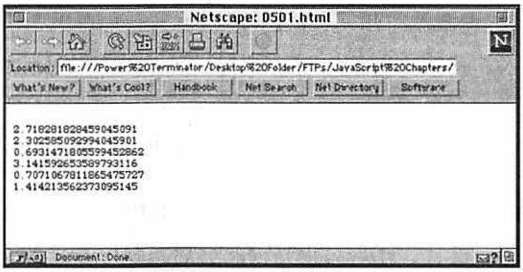
Math Properties

The `Math` object provides six properties. These properties hold fundamental constants—constants that are often used in mathematical functions. These properties follow:

- □ `E`. This property holds the value for Euler's constant, which is approximately 2.718.
- □ `LN10`. This property holds the natural logarithm of 10, which is approximately 2.302.
- □ `LN2`. This property holds the natural logarithm of 2, which is approximately 0.693.
- □ `PI`. The property PI is the constant that holds the ratio of a circle's circumference to its diameter. The value is approximately 3.1415.
- □ `SQRT1_2`. This property holds the square root of one half (1/2), approximately 0.707.
- □ `SQRT2`. This property holds the square root of 2, which is approximately 1.414.

Figure 6.2 shows all the values displayed using the following script:

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!--
with(Math) {
document.writeln(E)
document.writeln(LN10)
document.writeln(LN2)
document.writeln(PI)
document.writeln(SQRT1_2)
document.writeln(SQRT2)
}
// -->
</SCRIPT>
</PRE>
</BODY>
</HTML>
```



Location: file:///Power%20Terminator/Desktop%20Folder/FTPs/JavaScript%20Chapters/

```
2.718281828459045091
2.302585092994045901
0.6931471805599452862
3.141592653589793116
0.7071067811865475727
1.414213562373095145
```

Figure 6.2 *The* `Math` *properties.*

`Math` Methods

In addition to the properties built into the `Math` object, several methods perform calculations. These include the following:

- □ `abs(`*`number`*`)`. This method returns the absolute value of the number provided. If the number is zero (0) or positive, the method returns the number unchanged. If the number is negative, however, the method returns the number without the negative sign.

- □ `acos(`*`number`*`)`. This method returns the arc cosine of *`number`*. *`number`* should be from –1 to 1, and `acos()` returns the value in radians, which run from 0 to Pi radians. If *`number`* is outside the –1 to 1 range, the method returns 0.

- □ `asin(`*`number`*`)`. This method returns the arc sine of *`number`*. *`number`* should be from –1 to 1, and `asin()` returns the value in radians, which run from –Pi/2 to Pi/2 radians. If *`number`* is outside the –1 to 1 range then the method returns 0.

- □ `atan(`*`number`*`)`. This method returns the arc tangent of *`number`*. *`number`* should be the tangent of an angle, and `atan()` returns the value in radians, which run from –Pi/2 to Pi/2 radians. If *`number`* is outside the –1 to 1 range, the method returns 0.

- □ `ceil()`. This method returns the integer that is the next integer further away from zero than *`number`*. `ceil(32.5)`, for example, would become 33, and `ceil(-24.8)` would become 24.

- □ `cos(`*number*`)`. This method returns the cosine of an angle (*number*), which is represented in radians. The result will run from –1 to 1.

- □ `exp(`*number*`)`. This method returns *e* to the power of *number*.

- □ `floor(`*number*`)`. This method returns the integer closest to *number* toward 0. For example, 32.5 would become 32, and –24.8 would become –24.

- □ `log(`*number*`)`. This method returns the natural logarithm of the value *number*.

- □ `max(`*number1*, *number2*`)`. This method returns the greater of two numbers.

- □ `min(`*number1*, *number2*`)`. This method returns the lesser of two numbers. For example, `min(45, 23)` would return 23.

- □ `pow(`*base*,*exponent*`)`. This method returns the value of base to the power of exponent. For example, `pow(2,3)` would return 8.

- □ `random()`. This method is only available on X-platforms, which are essentially any UNIX operating systems. It returns a pseudo random number between 0 and 1.

- □ `round(`*number*`)`. This method rounds the value of *number* to the nearest integer. If the decimal portion of the number is .5, the number rounds up.

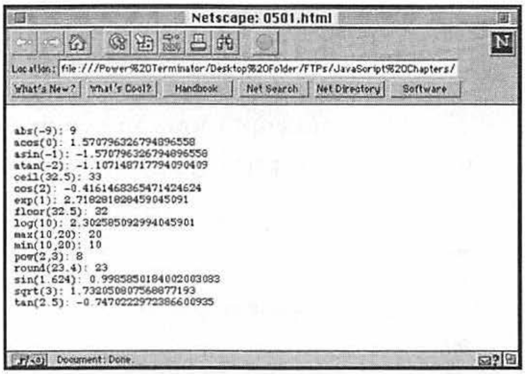
- □ `sin(`*number*`)`. This method returns the sine of an angle (*number*), which is represented in radians. The result will run from –1 to 1.

- □ `sqrt(`*number*`)`. This method returns the square root of the value of *number*. The value of `number` must be nonnegative; otherwise the function always returns 0. If you wish to deal with complex numbers, you have to use the `a + bi` notation where the real and imaginary components are separated.

□ tan(*number*). This method takes an angle in radians as its argument and returns the tangent to the angle.

The following script shows an example of each of these functions in use. The results provided are shown in Figure 6.3.

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!--
with(Math) {
document.writeln("abs(-9): " + abs(-9))
document.writeln("acos(0): " + acos(0))
document.writeln("asin(-1): " + asin(-1))
document.writeln("atan(-2): " + atan(-2))
document.writeln("ceil(32.5): " + ceil(32.5))
document.writeln("cos(2): " + cos(2))
document.writeln("exp(1): " + exp(1))
document.writeln("floor(32.5): " + floor(32.5))
document.writeln("log(10): " + log(10))
document.writeln("max(10,20): " + max(10,20))
document.writeln("min(10,20): " + min(10,20))
document.writeln("pow(2,3): " + pow(2,3))
//document.writeln("random(): " + random())
document.writeln("round(23.4): " + round(23.4))
document.writeln("sin(1.624): " + sin(1.624))
document.writeln("sqrt(3): " + sqrt(3))
document.writeln("tan(2.5): " + tan(2.5))
}
// -->
</SCRIPT>
</PRE>
</BODY>
</HTML>
```



```
abs(-9): 9
acos(0): 1.5707963267948966558
asin(-1): -1.5707963267948966558
atan(-2): -1.1071487177940904 09
ceil(32.5): 33
cos(2): -0.4161468365471424624
exp(1): 2.718281828459045091
floor(32.5): 32
log(10): 2.302585092994045901
max(10,20): 20
min(10,20): 10
pow(2,3): 8
round(23.4): 23
sin(1.624): 0.998585018400200 3083
sqrt(3): 1.7320508075688 77193
tan(2.5): -0.7470222972386600935
```

Figure 6.3 *Results of the* `Math` *methods.*

A simple example of one of the `Math` methods in action is shown in Figure 6.4. In this case, an array of rows is looped, each representing a value from 1 at the top to -1 at the bottom. As the script moves along the columns, it progressively moves through the radian measurements in 0.1 radian increments. If the value of sin at that point is less than the row value, a pound sign (#) is printed. The code that carries out this wave is as follows:

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!--
with(Math) {
for(var i = 0; i < 20; i ++) {
    for(var j = 0; j < 7; j += 0.1) {
        if((10 + sin(j)*10) > (20 - i)) {
            document.write("#")

        }
        else {
            document.write(" ")
        }
    }
    document.write("\n")
}
}
// -->
</SCRIPT>
</PRE>
</BODY>
</HTML>
```

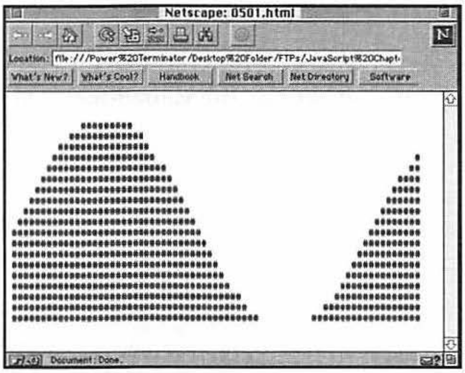


Figure 6.4 *A sine wave using JavaScript.*

NOTE The `Math` functions in JavaScript, when dealing with double-precision floating-point numbers, are subject to platform-dependent accuracy bugs when numbers get close to boundary conditions such as +/- Inf, and NaN (Not a Number, a condition found in Unix systems). What does this probably mean for you, the HTMLer who just wants a few numbers crunched? Stay away from JavaScript when completing your quantum mechanics homework. Otherwise, you should be fine.

Be aware of these problems when using the `Math` methods.

The `Date` Object

As with the `String` object, JavaScript does not have a native `Date` type, so it provides the programmer with an object that encapsulates information about a date and time and provides methods for handling this information. The `Date` object is similar to the Java `Date` object in that it stores the date as the number of milliseconds since January 1, 1970, 00:00:00. Because of this format, there is no way to store dates prior to January 1, 1970.

Date Constructors

To use Date methods, you must first create an instance of a date with the value you want. You can construct a Date object three different ways.

```
tempDate = new Date()
```

This constructor method creates a new date and stores the current date and time as the value for the object.

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

This constructor takes a string such as "January 22, 1996 06:32:00" and creates a date. Omit any of the time values, and they will be set to zero automatically.

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

This constructor uses comma-separated numbers instead of a string and sets the Date object accordingly. Again, you can omit any of the time values and zero will be substituted.

Date Methods

The Date object does not have any properties that can be set or accessed directly. Instead, it provides several methods that affect Date values.

The first group of methods enables you to set information about the date.

Table 6.1

Method	Range of Valid Values
setDate(*dayOfMonth*)	1–31
setHours(*hours*)	0–23
setMinutes(*minutes*)	0–59
setMonth(*month*)	0–11
setSeconds(*seconds*)	0–59
setTime(*milliseconds*)	0 and up
setYear(*year*)	1970 and up

The Date set methods enable you to control the value of Date objects after they are created. Each method takes an integer in the range specified in the preceding table. Most of the methods are straightforward, except for the setTime method. This method takes the time as the number of milliseconds after January 1, 1970, 00:00:00. Obviously, this is not a convenient notation to use. This method is intended for use in conjunction with the getTime method to set the time from an already established Date object.

In addition to the Date set methods, there are also several Date get methods for returning the values of different elements of a Date.

For all of the set functions, there exist identical get functions. These get functions return the integers that represent the values specified in the preceding table.

Table 6.2

Method	Return Value
getDate()	Returns the day of the month
getDay()	Returns the day of the week
getHours()	Returns the hour of the day
getMinutes()	Returns the minutes in the hour
getMonth()	Returns the month
getSeconds()	Returns the seconds in the minute
getTime()	Returns milliseconds since 1.1.1970
getTimezoneOffset	Returns the offset from GMT
getYear()	Returns the year—post 1970

There are two methods—getDay and getTimezoneOffset—that return values that cannot be set directly through the Date object. The getDay method returns the day of the week. Because this is determined by the other Date values, there is no need to set this number explicitly. The getTimezoneOffset method returns the offset from Greenwich Mean Time in minutes of the client's computer locale. This is information set by the operating system, not the programmer.

In addition to the set and get methods, the `Date` object has five other methods. Two of these methods—`parse()` and `UTC()`—are static and are implemented using the `Date.`*`method()`* syntax rather than being appended to an actual instance of an object.

□ `parse(`*`dateString`*`)`. This method parses the string representation of `Date` and returns the number of milliseconds from January 1, 1970, 00:00:00. The date is expected to be in local time. The `Timezone` offset of the computer will be used unless a GMT stamp is placed on the end of the string, for example, "`GMT-0330.`" Notice that the GMT offset is in hours and minutes (*hhmm*), not total minutes.

□ `UTC(`*`year`*`, `*`month`*`, `*`day`*` [, `*`hours`*`] [, `*`minutes`*`] [, `*`seconds`*`])`. This method returns the number of milliseconds since January 1, 1970, 00:00:00, based upon Greenwich Mean Time (GMT) or Universal Coordinate Time (UCT). Use this method if you have a date based upon GMT instead of local time.

There are also three methods for producing a string output of the date. These are

□ `toGMTString()`. This method returns the date in GMT using Internet conventions.

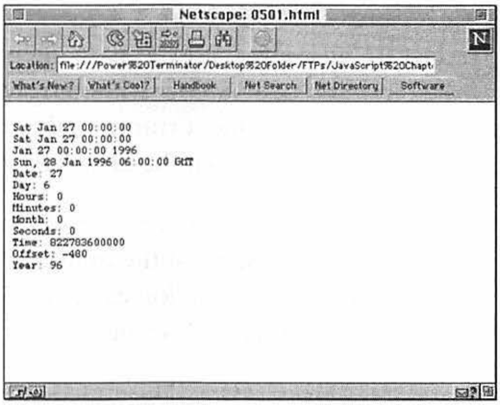
□ `toLocaleString()`. This method returns the date in local time, using the mm/dd/yy hh:mm:ss format.

□ `toString()`. This method returns the date in local time, using the Internet standard of Month, day year hh:mm:ss.

Figure 6.5 shows the get methods and `toString` methods for the `Date` object produced by the following source code.

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!--
today = new Date()
today.setTime(Date.parse("January 26, 1996"))
```

```
document.writeln(today)
document.writeln(today.toString())
document.writeln(today.toLocaleString())
document.writeln(today.toGMTString())
document.writeln("Date: "+today.getDate())
document.writeln("Day: "+today.getDay())
document.writeln("Hours: "+today.getHours())
document.writeln("Minutes: "+today.getMinutes())
document.writeln("Month: "+today.getMonth())
document.writeln("Seconds: "+today.getSeconds())
document.writeln("Time: "+today.getTime())
document.writeln("Offset: "+today.getTimezoneOffset())
document.writeln("Year: "+today.getYear())
// -->
</SCRIPT>
</PRE>
</BODY>
</HTML>
```



```
Sat Jan 27 00:00:00
Sat Jan 27 00:00:00
Jan 27 00:00:00 1996
Sun, 28 Jan 1996 06:00:00 GMT
Date: 27
Day: 6
Hours: 0
Minutes: 0
Month: 0
Seconds: 0
Time: 822783600000
Offset: -480
Year: 96
```

Figure 6.5 *Output from several of the* `Date` *methods.*

The eval, parseInt, parseFloat, escape, unEscape, and isNaN Functions

The final units of built-in functionality in JavaScript are the `eval`, `parseInt`, `parseFloat`, `escape`, `unEscape`, and `isNaN` functions. Unlike the objects discussed previously, these functions are standalone methods that are not attached to any object. Therefore, they can be called without reference to any one particular instance or static reference to an object, for example, `Date` or `Math`.

- □ `eval(`*`statement`*`)`. The *eval* function takes a string and returns the numerical equivalent of any operations represented by the string. For example, the following line

  ```
  result = eval("2+6/2");
  ```

  would return the number 5. This function is important in form elements such as user-entered text fields, where you are provided with a string representing the function you need to evaluate. You might, for example, have a spreadsheet script in which you need to evaluate arithmetic expressions in a cell represented by a text input field. You could then take the string from the text field and return the numerical equivalent.

- □ `parseInt(`*`numberString, radix`*`)`. This function returns the numerical equivalent of the integer string based upon the integer *`radix`* provided. *`radix`* represents the base of the number to convert to such as 10 for decimal, 8 for octal, and 16 for hexadecimal. Because `parseInt` expects an integer, it truncates any decimal portion of a number without rounding. If the method cannot parse the string, it truncates the number wherever it ended. If it cannot parse even the first character, it returns 0.

- □ `parseFloat(`*`floatString`*`)`. This function returns the floating-point equivalent of the number represented by *`floatString`*. The `parseInt` function can handle floating-point numbers in both decimal and scientific notation.

- □ `escape(`*`character`*`)`. This function returns the ASCII encoding of the argument in the ISO Latin-1 character set. For example, `escape("&")` returns `"%26"`.

- □ `unescape(`*`string`*`)`. This function returns the ASCII character for the encoded string argument. For example, `unEscape("%26")` returns `"&"`. Both `escape` and `unEscape` can be used to encode characters that would not normally be understood by a file type. In Chapter 8, these functions are used for the `document.cookie` property.

- □ `isNaN(`*`number`*`)`. This function is only available on Unix platforms and tells you whether a number is of type NaN, or Not a Number.

Summary

The built-in functions of the JavaScript language provide a useful set of features that programmers can use to develop powerful scripts quickly. Most programmers who work with today's programming languages have become accustomed to the kind of built-in support that these objects provide to JavaScript. The next two chapters cover the objects that are built into the client-side browser environment of Netscape. These include the document, window, history, and location objects, as well as the assorted member objects for each of these object classes. You need to understand the hierarchical relationship among all of these elements in order to fully realize fire-breathing, JavaScript-powered Web pages.

CHAPTER 7

Netscape Navigator Objects: The Document Object

After you get beyond the objects built into the JavaScript language, the Netscape Navigator program provides several objects that enable the scripter to interact with HTML documents. The main objects that fall into this category are the document and form objects. These objects enable the user to control and respond to events in the Navigator environment. In fact, the document and form objects are at the bottom of the Navigator object hierarchy, which is why they are the first and easiest objects that can be used in a JavaScript program without extensive knowledge of the Navigator hierarchy. You encountered the document object in the first chapter of this book, when its `write()` method was used to send text to display in the Navigator window.

Introducing the Navigator Object Hierarchy

The complete Netscape Navigator object hierarchy is shown in Figure 7.1, and depicts the relationship among objects provided by the Navigator environment. Many of these objects are dependent upon what the HTML source actually generates. For example, if your HTML file does not include any forms, then no form objects are present. If no links are created, then no link objects are present. In this way, the object hierarchy is very dependent upon the current environment.

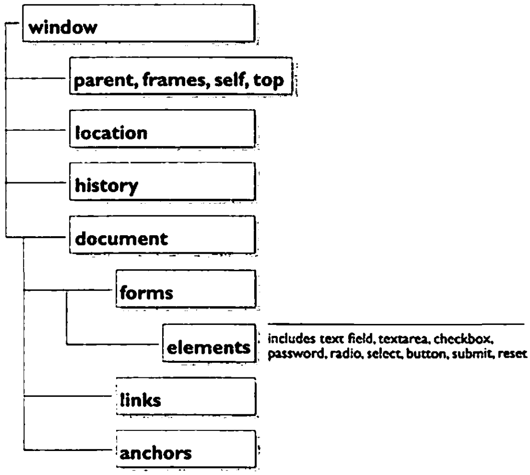

window
parent, frames, self, top
location
history
document
forms
elements
includes text field, textarea, checkbox, password, radio, select, button, submit, reset
links
anchors

Figure 7.1 *The Netscape Navigator object hierarchy.*

At the top of the hierarchy is the window object. This object is the encapsulation of the Netscape Navigator window in which a document is being displayed. In addition to its own window properties, this object also holds several other objects that can be considered members of the window. These objects include the following:

- □ Any frames that have been created
- □ The location and history objects that hold information about the current URL and URLs forward and backward in the link history
- □ The document or documents (if there is more than one frame) contained in the window/frames

The window object, along with the location and history objects, are covered in more detail in Chapter 8, "The Forms, Window, History, and Location Objects."

Introducing Window Object Properties

The window object's first property in the hierarchy includes several similar objects that all refer to windows or frames within windows that are being displayed by the Netscape Navigator program. The parent object is used when calling functions or object methods that reside in the window object, and are being called from a frame that exists as a child of the window. There are also two more named objects, which are used to refer to windows: Self objects refer to the current window, and top objects refer to the original window that was accessed when Netscape was first loaded. Finally, the window object includes each frame that occupies screen space. Therefore, if there are two frames, there will be two frame objects. All these objects are dependent on the HTML code that is passed to the Navigator program from the source file, whether it is code generated on-the-fly or static HTML tags.

Two objects that do not depend on the HTML code are the location and history objects. Although the content of these objects changes as the Web-browsing user moves through sites during a surfing session, their existence is static. The location object refers to the current URL of the source file creating the HTML source code. Because the Navigator 2.0 JavaScript specification does not implement the SRC= attribute for the <SCRIPT></SCRIPT> tag, the file URL for the script and the HTML document are the same. In the future, when the SRC= attribute is included in the <SCRIPT></SCRIPT> tag, this URL specified by this attribute will be the parent HTML document to which this object will refer. The history object holds information about the links a user has visited; these are the same links shown in Netscape's Go menu.

The document object is the final and most important object in the Navigator hierarchy. Its importance is seen in its immediate use in even the most rudimentary JavaScript examples, such as Hello, World!. You will undoubtedly need to use the document object if you want the end user to see the results of most things you do in your scripts. The document object has several objects as properties. The three most important of these are the following:

□ Link object

□ Anchor object

□ Form object

The link and anchor objects are dependent on the existence of links or anchors in the document's HTML code. Otherwise, they are not created. The form object is also dependent on the existence of <FORM></FORM>tags in your HTML code.

The link object refers to <A HREF=></A> tags that link an element, such as text or an image, to a specific URL. The link object is not useful on its own, but exists as a property of the document object itself. When links exist, they can be referenced as an array property of the document object. The anchor object refers to any <A NAME=></A> tag that exists in the HTML code. Like the link object, the anchor object exists as an array property that holds all the anchors in a document. Both of these objects are covered in more detail later in this chapter.

The form object is also included as a property of the document object, and like the link and anchor objects, exists as an array that holds each of the forms as a single element. For example, the first form on the page would be `document.forms[0]`, `document.forms[1]`, and so on. In addition, each of the form areas in a document has properties for each of the interface elements in the form, such as buttons and text entry fields. These elements can then be accessed through the document object. The forms property can be used to control the validation of forms before they are sent to a server-side CGI program or other protocol to be handled, thus saving much of the networking overhead that is presently a persistent element of much HTML interaction.

The Netscape Navigator object hierarchy is quite extensive, and the number of levels the programmer is expected to work through to use a specific object can be quite tedious. To ensure that the correct objects are being used, however, and to create as flexible a tool as possible, the object hierarchy becomes a useful device for implementing complicated scripts that rely on several frames or windows to carry out their tasks. Also, by using the hierarchical structure, JavaScript stays closer to the object-oriented design principles upon which it is based, and keeps the programmer thinking along these lines as well.

The Document Object

The document object as it was just presented exists in the middle of the Navigator object hierarchy, between the higher-level windows and frames, which are used to partition screen real estate, and the lower-level elements (that is, links and form elements) that together comprise the entire document presented to the end user. The document object's job is to bring all these elements together in one logical place and expose them to the programmers for use in their JavaScript code. This object-oriented approach to the encapsulation of HTML elements is central to JavaScript, and many of the objects are designed to allow interaction with different elements of the HTML language.

The HTML Equivalent of the Document Object

The document object actually refers to the body of the HTML document that is defined by the <BODY></BODY> tag. The body of an HTML document takes several optional attribute tags. The following code defines all these options:

```
<BODY
    BACKGROUND="background image"
    BGCOLOR="background color"
    FGCOLOR="foreground color"
    LINK="unfollowed link color"
    ALINK="activated link color"
    VLINK="followed link color"
    onLoad="event handler code"
    onUnLoad="event handler code">
</BODY>
```

All these options, except the background image, are included as properties of the document object and can be referenced by the document.*property* naming convention. In addition to the standard attributes, the <BODY> tag can also include the onLoad and onUnLoad event handlers that execute the specified code the same as all event handlers do when the action of loading or unloading the document occurs.

When creating HTML documents, it is important to take into account the browsers that do not support the JavaScript standard, or older versions of browsers that do. If you are looking for a specific color scheme, be sure to use the standard HTML <BODY> tag to define these preferences. Then, if you want to enhance the page, you can use the document object's properties to override these settings.

Understanding the order in which document attributes are assigned when using both the object properties and the HTML tags is important. In the following code, both the JavaScript code and the HTML tag set the background color. The document object property, however, is used when the page is actually displayed.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
document.bgColor = "blue"
function change() {
    document.bgColor = "red"
}
// -->
</SCRIPT>
</HEAD>
<BODY BGCOLOR="white">
<FORM>
<INPUT TYPE="button" NAME="red" VALUE="RED" onClick="change()">
</FORM>
</BODY>
</HTML>
```

Two things are important to note. First, the document property setting overrides any HTML tag settings in the body statement. It does not matter that the assignment of the color blue to the `document.bgColor` property occurs in the <HEAD></HEAD> before the body statement. It carries through in this case. Second, the background color can be updated dynamically. Even though the screen display has been completed, the change in the `bgColor` property that occurs as a result of the `change()` function call in the red button's `onClick` event handler causes the screen's `bgColor` property to update immediately. (`bgColor` is the *only* property that does update right away; the various link color properties are updated internally, but until the link is activated, the changes do not take effect.) Try the following code that changes several of the document properties to see how changing the document property values changes the colors of the links:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
document.bgColor = "blue"
document.vlinkColor = "white"
document.linkColor = "yellow"
document.alinkColor = "red"
function change() {
    document.bgColor = "red"
    document.vlinkColor = "blue"
    document.linkColor = "green"
    document.alinkColor = "blue"
}
// -->
</SCRIPT>
</HEAD>
<BODY BGCOLOR="white">
<A HREF="http://www.purdue.edu/">Purdue University</A>
<A HREF="http://www.ac.com/">Andersen Consulting</A>
<FORM>
<INPUT TYPE="button" NAME="red" VALUE="red" onClick="change()">
</FORM>
</BODY>
</HTML>
```

NOTE Be sure to note that the default behavior of mixing JavaScript and HTML source code is not at all obvious or straightforward. This is due mainly to the order in which items are evaluated and which elements of the display are updated at what time. Be sure to confirm the behavior of a particular script; don't automatically assume that it will act as you intend. Because JavaScript and HTML can be intertwined quite freely, it is difficult to define exactly how the two will behave in each instance. The Netscape 2.0 implementation of JavaScript still has a few quirks that should be fixed in the next release, slated as Version 2.1.

The next two sections detail the properties, methods, and event handlers that are part of the document object as defined in the Netscape Navigator 2.0 specification.

The Document Object Properties

The properties of the document object include the following:

□ `alinkColor`

□ `anchors`

□ `bgColor`

□ `cookie`

□ `fgColor`

□ `forms`

□ `lastModified`

□ `linkColor`

□ `links`

□ location

□ referrer

□ title

□ vlinkColor

This list of properties can be divided into two groups, with properties such as alinkColor and bgColor falling in an Attributes group, and properties such as forms and links falling in an Elements group. The properties in the first group exist regardless of what is contained in the actual <BODY> tag or created by document.write() methods. These properties hold information about attributes that always exist, such as the background color (bgColor) or the color of a visited link (vlinkColor). These properties are nearly equivalent to the attributes that can be set in the standard HTML body tag, and therefore are considered the document object's attribute properties. The properties that fall into the second group are dependent on the existence of their respective elements in the document's <BODY> tag. These include arrays such as links and forms and the cookie string, which is used to hold data on the client machine between browser sessions. (Cookies are discussed in detail in the "Document Element Properites" section.) Because these properties are created by the elements that make up the body of the document, they will be referred to as the document object element properties.

The Document Attribute Properties

The document attribute properties include the following variables:

□ alinkColor

□ bgColor

□ fgColor

□ lastModified

□ linkColor

□ location

- □ title
- □ vlinkColor

Of all these properties, only the location and lastModified variables are not definable in the <BODY></BODY> HTML tag. Each of the other properties directly relates to one of the attribute tags shown in the previous section.

The Color Properties

The color properties include the alinkColor, bgColor, fgColor, linkColor, and vlinkColor properties. These properties control the color formatting of the page in reference to the condition of the HREF links, if any in the document, and the background and foreground colors. The color value provided is a string that is of the form

```
alinkColor = "#RRGGBB"
```

The RR, GG, and BB represent the red, green, and blue channels that make up the color spectrum and run in value from 0 to 255, represented in hexadecimal values. For example, the basic colors are

red = “#FF0000”

blue = “#0000FF”

green = “#00FF00”

white = “#FFFFFF”

black = “#000000”

By mixing different values for the red, green, and blue channels, you can control the exact color. Of course, counting in hex is not at all intuitive, so JavaScript and the HTML code can accept names for colors. These names are listed in Appendix A and provide almost any combination you could reasonably expect. If you are trying to match an exact RGB value, it might be easier to use the exact hexadecimal numbers.

NOTE **Adobe Photoshop can "sample" the colors in an image or on a color palette and translate those colors into decimal RGB values. And if you don't feel like converting decimal values to their hexadecimal equivalents, a number of Web sites are available that can do the conversion for you. One straightforward site that previews the RGB color you submit for translation is the BgColor Server (http://www.webedge.com/bgServer.fcgi).**

□ `alinkColor`. This property controls the color of an active link. This is the color that a link temporarily changes to as the user presses the mouse button while the cursor is over the link. Notice that when selecting buttons or links, it is not the lone act of pressing the mouse down that selects the item. Rather, it is the combination of pressing ***and*** releasing the mouse button over the object that selects it. If you move the mouse away from the object before releasing the mouse button, it will not be selected.

□ `linkColor`. This property controls the color of the standard HTML link created in an <A HREF="*URL*">*text*</A> tag. This color attribute is not automatically updated if it is changed after the document has been displayed. As soon as a link in the document is selected, however, all the link colors are updated to show any new, dynamically updated `linkColor` setting.

□ `vlinkColor`. This property is similar to the `linkColor` property, except this color is used for links that the Navigator has stored as being previously visited. Like the `linkColor` property, and unlike the bgColor property, this property is not reflected in the displayed document until after a link has been selected.

- □ bgColor. This property holds the value for the background color of the document. When changed, this property is immediately updated in the display.
- □ fgColor. This property holds the value for the foreground text color in the display. It does not appear to be changeable after the color is set and the document has been read by the browser.

The title, location, and lastModified Properties

Strictly speaking, these two properties are not members of the <BODY></BODY> tag, but are considered attributes of the document in general.

- □ title. This property holds the value of the title of the document as created in the <TITLE>*document title*</TITLE> tag. This title is displayed in the title bar of the browser window, and is the name given to the URL in the Bookmarks menu if the user adds it as a bookmark. This value cannot be changed after the document has been created.
- □ location. This property holds the full URL of the file holding the document source code. Because the Netscape 2.0 JavaScript specification has not implemented the capability to import JavaScript files by using the SRC= attribute in the <SCRIPT></SCRIPT> tag, the URL provided is the file from which the document source code comes.
- □ lastModified. This property holds the date the file was last modified. In conjunction with the document cookie, this could be used to inform users that modifications have been made since they last visited the site. The cookie property will be discussed in more detail in the next section.

The Document Element Properties

The document properties that reflect the elements present in the HTML document are the following:

- □ anchors
- □ cookie
- □ forms
- □ links

The anchors, forms, and links properties are arrays that have an entry for every element of each type existing in the <BODY></BODY> portion of the HTML code. The JavaScript programmer can then use these arrays to refer to the individual element objects by using index values in the array. The cookie property is a string representing the file MagicCookie. This file holds what are known as cookies, which enable HTML pages to store a limited amount of information specific to themselves on the client machine. For example, a Web site could use a cookie file on your Mac to personalize the greeting its home page gives you each time you log in: "Welcome, Dave Thomas. It's been 7 days since your last visit to MurkyWeb," or some such thing. Fox Mulder believes the "Truth is out there," and we concur—it's probably in the cookie files.

The anchors Document Property

The anchors document property contains all the anchors in a document in an array structure. The order of the items in the anchors array is dependent on the order in which the anchors appear in the document. Unlike the standard array structure created with the makeArray() function created in Chapter 5, "Using and Creating Objects in JavaScript," the anchors, and all the element properties of the document object, begin indexing at zero (0) and continue through one less than the total number of objects (n–1). Therefore, the first anchor in the document is at document.anchors[0], the second at document.anchors[1], and so on. Like the standard array, the total number of elements in the array is held in the length property of the array, and the total number of anchors in a document would be document.anchors.length. The following code prints out the number of anchors in the document.

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="white">
This is a test:<P>
<A NAME="ANCHOR 1" HREF="http://www.purdue.edu">Purdue University
➥</A><P>
<A NAME="ANCHOR 2">CompuServe</A><P>
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!--
document.writeln("This document has " + document.anchors.length + "
➥anchors.")
// -->
</SCRIPT>
</PRE>
</BODY>
</HTML>
```

Notice that anchors also refer to HREF links. The anchors property refers to the existence of the NAME= attribute in the <A> </A> tag that provides a hypertext target for a URL to move within the page.

The links Document Property

The links document property is quite similar to the anchors property, except that instead of looking at all the instances of the NAME= attribute in <A></A> tags, the links property lists all the <A></A> tags with HREF= attributes. The links array begins indexing at zero (0) and goes through one less than the total number of objects link elements (n–1). Therefore, the first link in the document is at document.links[0], the second at document.links[1], and so on. The total number of anchors in a document would be document.links.length. The following code is identical to the preceding example, except this time it counts only the links in the document.

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="white">
This is a test:<P>
<A NAME="ANCHOR 1" HREF="http://www.purdue.edu">Purdue University
➥</A><P>
<A NAME="ANCHOR 2">CompuServe</A><P>
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!--
document.writeln("This document has " + document.links.length +
" links.")
// -->
</SCRIPT>
</PRE>
</BODY>
</HTML>
```

The forms Document Property

The forms document property is an array of objects that corresponds to all the named forms in a document. These forms are indicated by using the <FORM></FORM> tag, and can have several additional elements, depending on which form input elements are included with them. The following code, like the two previous examples, counts the number of forms in a document.

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="white">
This is a test:<P>
<FORM NAME="FORM 1">
<INPUT TYPE="text" NAME="form1Text" VALUE="hello" SIZE=10>
<INPUT TYPE="button" NAME="form1Button" VALUE="test">
</FORM>
This is the second form:
<FORM>
<SELECT NAME="form2Select" SIZE=3 MULTIPLE>
<OPTION>Apple Computer
<OPTION>Netscape
<OPTION>Sun Microsystems
```

```
</SELECT>
</FORM>
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!--
document.writeln("document has " + document.forms.length + " forms")
// -->
</SCRIPT>
</PRE>
</BODY>
</HTML>
```

Chapter 8 details the form object and how it can be used in JavaScript programs.

The `cookie` Document Property

The `cookie` property is perhaps one of the most interesting, yet complicated, features of the Netscape environment. A *cookie* is an entry in a text file called MagicCookie that resides in the Netscape preferences folder. This file typically has been used by server-side CGI scripts to keep information on the client machine for whenever they might visit again. JavaScript enables the programmer access to this file to store recurring information. It is important to recognize two things about cookies. First, the Netscape environment can create only 300 cookies in total, thus limiting the capability of a site to swamp users' hard disks with information they do not necessarily want. In addition, each of these cookies is limited to 4 KB in size. This effectively limits the file to about 1.2 MB. Moreover, a single site may have only 20 cookie entries, and sites are limited to accessing cookie entries to those that match the same domain. The full cookie entry appears as the following:

```
NAME=VALUE; expires=DATE; path=PATH; domain=DOMAIN_NAME; secure
```

The name is the name given to the cookie as an identifier, and the value is the value of the cookie itself. The expires date indicates how long the cookie should remain on the client. If this is not provided, it will only remain for the current session. The entry format for the expires entry is

```
Wdy, DD-Mon-YY HH:MM:SS GMT
```

The path and domain entries are there for specifying which URLs can access the cookie entry. If the URL of the document attempting to access the cookie does not match the domain name and path, it is not allowed to view the cookie. If two sites are in the same domain (for example catal.arch.cam.ac.uk and juno.arch.cam.ac.uk) and a common domain is given in the cookie (arch.cam.ac.uk), both will have access to the cookie. Typically, at least three entries separated by two periods are needed to qualify the domain name. If the site is one of the major types of "com," "edu," "net," "org," "gov," "mil," or "int," then the domain needs only two periods; any others need three. A site can specify a domain entry only if it is in that domain. If the path or domain is not provided, they default to the exact URL values of the document creating the cookie. The secure keyword is used when you only want the cookie sent if there is a secure connection. If this is not specified, the cookie will be transferable over nonsecure links.

The `cookie` property of the document object is a string that holds values for the cookie entry of the document. This value is altered by adding string expressions to the cookie. For example, you could have a function that places a string and an expiration date in that document's cookie:

```
function sookie(name, value, expires) {
    document.cookie = name + value + ";" + "expires=" + expires + ";"
}
```

Several functions have been developed and are in the public domain for working with cookies. The following, written by Bill Dortch, enable the user to set and examine cookies in a very flexible fashion. The following functions could be placed at the beginning of a file and used throughout your code to work with cookies. The cookie set does not use any white space, semicolon, or comma characters, so these must be encoded. The following `cookie` functions use the `escape()` and `unEscape()` functions built into JavaScript to convert standard characters to their ISO Latin-1 equivalents such as %26 for &.

This first function is used by the others as an internal method for accessing the items in the cookie.

```
function gookieVal (offset) {
  var endstr = document.cookie.indexOf (";", offset)
  if (endstr == -1)
    endstr = document.cookie.length
  return unescape(document.cookie.substring(offset, endstr))
}
```

This function returns the value of the cookie element specified by the string name. If there is no entry by that name, then it returns null.

```
function Gookie (name) {
  var arg = name + "="
  var alen = arg.length
  var clen = document.cookie.length
  var i = 0
  while (i < clen) {
    var j = i + alen
    if (document.cookie.substring(i, j) == arg)
      return gookieVal (j)
    i = document.cookie.indexOf(" ", i) + 1
    if (i == 0) break
  }
  return null
}
```

The following function can be used to create cookie entries. Only the first two arguments are required; any additional arguments are optional. If you decide not to use a cookie argument, such as `expires`, and provide an argument that occurs later in the list, such as `secure`, you must use `null` as a placeholder for the skipped argument(s). If the arguments being skipped come at the end of the argument list with none provided after, then you do not need to use `null` anywhere in the argument.

```
//  Function to create or update a cookie.
//    name - String object object containing the cookie name.
//    value - String object containing the cookie value.  May contain
//      any valid string characters.
//   [expires] - Date object containing the expiration data of the
➥cookie.  If
//     omitted or null, expires the cookie at the end of the current
➥session.
//   [path] - String object indicating the path for which the cookie
➥is valid.
//     If omitted or null, uses the path of the calling document.
//   [domain] - String object indicating the domain for which the
➥cookie is
//     valid.  If omitted or null, uses the domain of the calling
➥document.
//   [secure] - Boolean (true/false) value indicating whether cookie
➥transmission
//     requires a secure channel (HTTPS).
//
//  The first two parameters are required.  The others, if supplied,
➥must
//  be passed in the order listed above.  To omit an unused optional
➥field,
//  use null as a placeholder.  For example, to call SetCookie using
➥name,
//  value and path, you would code:
//
//     SetCookie ("myCookieName", "myCookieValue", null, "/")
//
//  Note that trailing omitted parameters do not require a
➥placeholder.
//
//  To set a secure cookie for path "/myPath", that expires after the
//  current session, you might code:
//
//     SetCookie (myCookieVar, cookieValueVar, null, "/myPath",
➥null, true)
//
function Sookie (name, value) {
  var argv = Sookie.arguments
  var argc = Sookie.arguments.length
  var expires = (argc > 2) ? argv[2] : null
  var path = (argc > 3) ? argv[3] : null
  var domain = (argc > 4) ? argv[4] : null
  var secure = (argc > 5) ? argv[5] : false
  document.cookie = name + "=" + escape (value) +
    ((expires == null) ? "" : ("; expires=" + expires.toGMTString()))
➥+
    ((path == null) ? "" : ("; path=" + path)) +
    ((domain == null) ? "" : ("; domain=" + domain)) +
    ((secure == true) ? "; secure" : "")
}
```

The following function enables the user to delete a cookie by name.

```
function DeleteCookie (name) {
  var exp = new Date()
  exp.setTime (exp.getTime() - 1);  // This cookie is history
  var cval = Gookie (name)
  document.cookie = name + "=" + cval + "; expires=" +
➥exp.toGMTString()
}
```

A possible scenario for using the cookie is as follows. When a page is first loaded, it first checks to see whether its cookie exists. If it does, it then uses the information stored in the cookie to set up the page to the same state it was in when the user left it. If the cookie function returns `null`, then the page knows that the user is there for the first time or the cookie has expired. In this case, it could provide a default or novice page. When the page is completed, for example on an `unLoad` call, the cookie could be updated with the last state of the page before exit. As you can see, the cookie provides a powerful means for creating dynamic, personalized documents for the Web.

The Document Object Methods

Five document methods exist that the programmer can use to control certain aspects of document objects. The methods are as follows:

- □ `clear()`
- □ `close()`
- □ `open()`
- □ `write()`
- □ `writeln()`

The last two methods, `write()` and `writeln()`, should already be somewhat familiar to you, because they have already been used extensively to create the output of many example HTML pages. The `clear()`, `close()`, and `open()` methods, however, probably are new to you, and provide a powerful means for creating documents from scratch.

The document.write() and document.writeln() Methods

The only difference between document.write() and document.writeln() is that the writeln() method inserts a newline character at the end of the line being displayed.

In most cases, the write() and writeln() methods are being sent to a document that is being interpreted as HTML text. Of course, in HTML, newline characters are ignored, and it is the line break tags such as <P> that designate when a new line should be created. Two tags exist that indicate to the browser that the text has been preformatted and that spaces and new lines should be kept intact. These are the <PRE></PRE> tags and the <XMP></XMP> tags. Using these tags enables the output from the write() and writeln() methods to be displayed correctly on-screen. Of course, the document to which the write() and writeln() methods are sending their values does not have to be of the text/html MIME type. Instead, document objects of the following type can be written to:

□ text/html

□ text/plain

□ image/gif

□ image/jpeg

□ image/xbm

□ x-world/*plug-in*

If you know the specification for a file type of GIF or JPEG format, you can use the write() method to dynamically create your own image document. The plug-in type is any plug-in that Netscape supports. You could, for example, create a VRML world on-the-fly that depends on user input for its construction.

The open(), close(), and clear() Document Methods

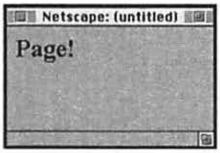
These three methods, open(), close(), and clear(), enable the programmer to create new documents, close documents, and clear documents from windows and frames. When document.write() is called, its default behavior is to write to the current document. If there is no document for the current window, then the write() method opens a new document of type text/html and writes to this document. Of course, in the default window of the Netscape browser, there is already a document open—the file in which the JavaScript is embedded. For this reason, there is no need to open a new document for output. As you will see in the next chapter, however, it is possible to open a window without a document. In this case, document.open() is required to begin the text stream that the browser interprets as the HTML file. After you have finished writing to the document, use the document.close() method to close the stream and display the file.

NOTE JavaScript and Netscape can become unstable if you call document.close() in the middle of a document's loading process. Thus, using document.close() is not recommended for use in the main window before all the HTML code has been displayed, because scripts typically appear between surrounding <HTML> </HTML> tags. However, you can call document.close() safely from an event handler because these are not evaluated until after the main document stream has closed anyway.

The following code opens a new browser window without a document. It then proceeds to send several documents to the window that create a flashing sign effect. Notice that each time the for loop is entered, it opens a new document in the msgWindow and closes it before it leaves.

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="white">
```

```
<SCRIPT LANGUAGE="JavaScript">
<!--
/**
 * this is used to create arrays in JavaScript
 */
function makeArray(n) {
    this.length = n
    for(var i = 1; i <= n; i++)
        this[i] = 0
    return this
}

text = new makeArray(5)
text[1] = "Hello, World!"
text[2] = "Welcome"
text[3] = "To"
text[4] = "My"
text[5] = "Page!"

// This creates the new window with no toolbars and sets the width
➥and height
msgWindow = window.open("","Hello",
➥"toolbar=no,width=200,height=100")

//this loop sends the different documents based upon the text array
➥to the msgWindow
```



Page!

Figure 7.2 *Creating new documents in a window.*

```
for(var j = 1; j <= text.length; j++) {
msgWindow.document.open()
msgWindow.document.write("<H1>" + text[j] + "<H1>")
msgWindow.document.close()
for(i=0;i<10000; i++){}
}

// -->
</SCRIPT>
</BODY>
</HTML>
```

Here is a breakdown of each of these methods:

- □ `open("`*`mimeType`*`")`. This method is used to open a new document stream. You can optionally specify what MIME type the document should be by providing the `mimeType` argument. The browser will then assume that the data being sent to it from the `write()` or `writeln()` methods are of that format. The list of valid MIME types are:
  - □ text/html
  - □ text/plain
  - □ image/gif
  - □ image/jpeg
  - □ image/xbm
  - □ x-world/*plug-in*
- □ You can use the `write()` method to create any of the listed-types that Netscape can interpret itself or through one of its plug-ins. The plug-in type is any plug-in that Netscape supports. You could, for example, create a VRML world on-the-fly. If no MIME type is given, then the default, text/html, is chosen.
- □ `close()`. This is the method used to close the document stream created with the `open()` method. If you want to create more than one document object, as in the flashing sign example, you will need to call `close()` before opening another document.
- □ `clear()`. This method clears the contents of the open document and empties the contents of the document window. This functions similarly to closing and then opening a new document in a window, as in the flashing sign example from the `open()` method definition.

The Document Object Event Handlers

The document object has two event handlers:

- □ `onLoad=`*`statement`*. This event handler executes the statement when the page is loaded. A useful action for an `onLoad` event handler is checking the value of a cookie when the page is first loaded and assigning it to a JavaScript variable that can then be used throughout the JavaScript code.

- □ `onUnLoad=`*`statement`*. This event handler causes the statement or statements to be executed when the document is unloaded. This event could be used to cause the page to update the persistent cookie before it is unloaded.

Summary

This chapter has provided the basis for using the document object in your JavaScript code. The document object is the central object in the Navigator object hierarchy, and provides many built-in functions for creating dynamic Web pages. Instead of depending on static HTML tags, you can generate documents on-the-fly. The next chapter includes the final sections of the JavaScript language we will cover. These are the form object, the window object, the location object, and the history object. They can be used both to control the hypertext links in your document and enable you to respond to user input through form elements.

CHAPTER 8

The Forms, Window, History, and Location Objects

By now you have been presented with the core of JavaScript programming. You have covered the foundations of the language, and learned about all the individual syntactical elements and how to use them in concert to create a functioning JavaScript program. In addition, the basic objects provided with the language have been covered, including the String, Math, and Date objects. Finally, the object central to JavaScript's impact on Netscape's dynamic behavior—the document—was presented in the last chapter.

Of course, with all these tools at your disposal, there are still several more topics to discuss before wrapping up the JavaScript language.

In the discussion of the document object, one property was given only cursory treatment: forms. Forms enable the client user to provide input to JavaScript programs, which can then be used to dynamically alter the behavior of a document, calculate a total, or even open a window to a new URL. In addition to forms, there are three other Navigator objects that exist at the same level as or above the document object:

□ The location object

□ The history object

□ The window object (above the document object)

The window object enables you to control the size and location of a document and provides access to dialog boxes and frames in addition to the standard Navigator window. The location and history objects encapsulate link information that provides JavaScript programs with the capability to react to information about the user's journey to his page, and about the user's Internet origins. By using these objects, the JavaScript programmer can enhance the basic document object with a diverse and dynamic range of interaction with the end user.

Using the Forms Object

Perhaps the most immediately useful aspect of the document object is its capability to access forms contained in HTML documents. The forms object enables JavaScript programs to interact with the different form elements manipulated by the user. In addition, it is also possible to access forms in a page by name.

The forms object is an encapsulation of the HTML codes that are used to create forms for HTML documents. The structure of the FORM tag is

```
<FORM
   NAME="form name"
   TARGET="target window"
   ACTION="form handling URL"
   METHOD=GET|POST
   ENCTYPE="encoding type"
   [onSubmit="JavaScript code"]>
</FORM>
```

The FORM tag is used to declare several attributes of the individual form:

- □ NAME is used to store the name of the form itself. It can be used to access the forms object in JavaScript code.
- □ TARGET is used to specify which window the response of form submission should go to. You can also designate a named frame within a<FRAMESET> tag as the target of the form response. In addition to these standard targets, you can specify special frames named _top, _parent, _self, and _blank. The targets enable you to specify targets that correspond to the JavaScript Navigator window hierarchy.

- □ ACTION is used to specify the URL of the program that will accept the data sent by the form. This is typically some type of CGI script on the server machine in the cgi-bin directory.
- □ METHOD has two possible values: GET or POST. By specifying GET, the form information is appended to the end of the URL, which is normally passed to a CGI (Common Gateway Interface) application in the 'srch' Apple Event parameter, better known as the "search args" parameter. The POST method places the form data in the body of the HTTP request from the browser, rather than appending it to the URL. The CGI application receives POST-method form data in the "post args" Apple Event parameter. Forms using the POST method can pass much more data than forms using the GET method.
- □ ENCTYPE indicates how the form is encoded. The two possible values are "application/x-www-form-urlencoded" or "multipart/form-data." A form's encoding indicates how user-entered form data is to be formatted in the HTML document returned if the POST method is used.

Properties

As with many JavaScript objects that reflect HTML tags, the list of attributes that follows are stored in the form object's property list. The form object properties include

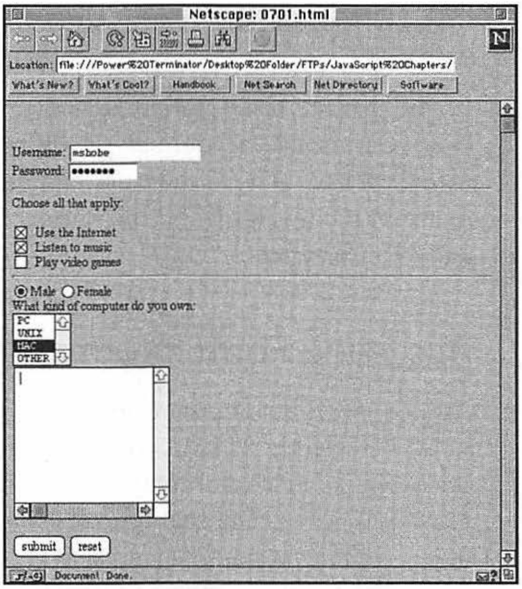
- □ `action`
- □ `elements`
- □ `encoding`
- □ `method`
- □ `target`

All these properties, except elements, reflect the state of the FORM tag's corresponding attributes. For example, the `action` property stores the value of the ACTION attribute.

The encoding property is unlike the other form properties in that it does not store the individual attributes of the form HTML tag. Instead, it is an array that stores the values of the individual form elements inside the FORM declaration. These elements can include buttons, text areas, check boxes and other features. Form elements are created in a FORM tag as follows:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<PRE>
<SCRIPT LANGUAGE="JavaScript">
<!--
// -->
</SCRIPT>
</PRE>
<FORM NAME="test">
Username:
<INPUT TYPE="text" NAME="text1" VALUE="" SIZE=20><BR>
Password:
<INPUT TYPE="password" NAME="password1" VALUE="" SIZE=10><BR>
<HR>
Choose all that apply:<P>
<INPUT TYPE="checkbox" NAME="checkbox1" VALUE="internet">
Use the Internet<BR>
<INPUT TYPE="checkbox" NAME="checkbox2" VALUE="music">
Listen to music<BR>
<INPUT TYPE="checkbox" NAME="checkbox3" VALUE="games">
Play video games<BR>
<HR>
<INPUT TYPE="radio" NAME="Sex" VALUE="Male">Male
<INPUT TYPE="radio" NAME="Sex" VALUE="Female">Female
<BR>
What kind of computer do you own:<BR>
<SELECT NAME="select1" MULTIPLE>
<OPTION>PC
<OPTION>UNIX
<OPTION>MAC
<OPTION>OTHER
</SELECT><BR>
<TEXTAREA NAME="textarea1" ROWS=10 COLS=20>comments
</TEXTAREA><P>
<INPUT TYPE="submit" NAME="submit1" VALUE="submit">
<INPUT TYPE="reset"  NAME="reset1" VALUE="reset">
</FORM>
</BODY>
</HTML>
```

These form elements then reside at `test.elements[0]`, `test.elements[1]`, and so on. If you access the form through the document object as the first form in the HTML code, then you can also access the individual elements by using `document.forms[0].elements[0]`, `document.forms[0].elements[1]`, and so on. Figure 8.1 shows the resulting form.



Location: file:///Power%20Terminator/Desktop%20Folder/FTPs/JavaScript%20Chapters/

Username: msbobe
Password: •••••••

Choose all that apply:

☒ Use the Internet
☒ Listen to music
☐ Play video games

● Male ○ Female
What kind of computer do you own:
PC
UNIX
MAC
OTHER

Figure 8.1 *The example form page.*

In the next section, the properties, methods, and event handlers for each of the form elements will be discussed.

Forms Object Methods

The only method available for the forms object is the `submit()` method. Invoking this method is exactly like the user pressing the submit button on the form. This can be used to automatically send the form information after some action has occurred in a JavaScript program. If you want to cause the test form shown above to be submitted, you can either call

```
document.test.submit()
```

or

```
document.forms[0].submit()
```

NOTE The `submit()` method has parentheses following it. It is a common mistake to forget the parentheses when calling a method. This call is in fact a reference to a property, which in the case of the forms object does not exist, and causes an error.

To say that the `submit()` method is the only forms object method does not take into account all the different methods that each element will have. The next section discusses each of the types of elements in the test form and their respective methods, properties, and event handlers.

Accessing Form Information

As mentioned before, there are several ways to access the forms objects created by HTML code in a document page. Because the document's HTML form tags are what create the forms in the first place, it is only fitting that forms be a property of the document object itself. In a similar manner to other document properties such as `links` and `anchors`, you can access the forms objects through the `forms` array. You can access the test form presented in the previous example with the following command:

```
document.forms[0]
```

If there are multiple forms in a document, you can access them in the order in which they occur by increasing the index of the array:

```
document.forms[1]
document.forms[2]
...
```

NOTE Remember that the array indexing of objects provided by the Netscape and JavaScript environments goes from zero to the number of objects minus one (0 to *n*–1). This is unlike the arrays that have been created with the `makeArray()` function in JavaScript you were shown in Chapter 4, which indexes from one to the number of objects in the array (1 to *n*).

In addition to accessing the forms objects that reside in the HTML document through the document `forms` array, it is also possible to refer to the forms objects by name if the name is provided in the FORM tag attributes. For example, a reference to the test form above by name appears as follows:

```
document.test
```

Notice that the test form must be accessed as a member property of the document object. What the document object essentially provides is two ways of accessing a forms object: through either its name or its array index.

NOTE To refer to a form, the form must already have been created in the document. The following code, which refers to the form below it in the document, causes a runtime error to be called:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
document.test.elements[0].value = "Hello, World!"
// -->
</SCRIPT>
</HEAD>
<BODY>
<PRE>
</PRE>
<FORM NAME="test">
Username:
<INPUT TYPE="text" NAME="text1" VALUE="" SIZE=20><BR>
...
<INPUT TYPE="reset"  NAME="reset1" VALUE="reset">
```

continues

```
</FORM>
</BODY>
</HTML>
```

The following code accesses the form successfully:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<PRE>
</PRE>
<FORM NAME="test">
Username:
<INPUT TYPE="text" NAME="text1" VALUE="" SIZE=20><BR>
...
<INPUT TYPE="reset"  NAME="reset1" VALUE="reset">
</FORM>
<SCRIPT LANGUAGE="JavaScript">
<!--
document.test.elements[0].value = "Hello, World!"
// -->
</SCRIPT>
</BODY>
</HTML>
```

The only change is that the reference to the form occurs after the form has been created in the document. Because event handlers are called only after the entire document has been loaded, you can place the call to the form in the <HEAD></HEAD> of the document in a function, which is called later by an event handler.

In addition to the placement of the references to forms, there are two additional important items to mention: changing the value of form items and form persistence. Form *persistence* means that when the users or programmer places data into a form element, Netscape retains that information, and if the page is reloaded during the same session (that is, without Netscape being closed), then the form elements will retain their same values.

Perhaps you noticed in the above example that the JavaScript statement accesses the value statement of the form's text element. When this value changes, it appears in the text box itself. This enables you to change the data in these boxes dynamically. For example, you can create a marquee to display a scrolling line of text. One way to do this is to create an array that holds all the characters (including blank spaces) for the text, and cycle through a portion of the characters you want to show at the same time on the screen. The following code creates such a marquee:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--

var buffer = "                                        "
var text = "Welcome to my Web Page, I hope you enjoy yourself!"
var marqueeString = buffer + text + buffer
var marqueeRunning= false
var timeoutID = null
var position = 0
var sLength = marqueeString.length

function stopMarquee() {
    if(marqueeRunning)
        clearTimeout(timeoutID)
    marqueeRunning = false
}

function startMarquee() {
    stopMarquee()
    updateMarquee()
}

function updateMarquee() {
        document.marquee.elements[0].value =
➥marqueeString.substring(position++%sLength,position+39%sLength)
        timeoutID = setTimeout("updateMarquee()", 50)
        marqueeRunning = true
}

// -->
</SCRIPT>
</HEAD>
<BODY onLoad="startMarquee()" onUnLoad="stopMarquee()">
<PRE>
</PRE>
<FORM NAME="marquee">
<INPUT TYPE="text" NAME="textBox" VALUE="" SIZE=40><BR>
</FORM>
</BODY>
</HTML>
```

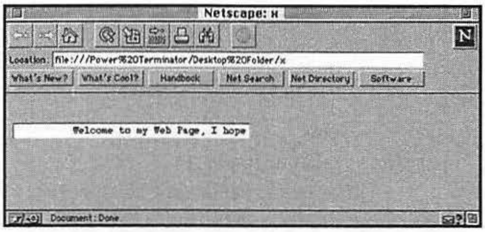
Notice the use of the `timeout()` method that causes the window to be updated every twentieth of a second. This method is part of the window object, which is presented later in the chapter. The most important line is

```
document.marquee.elements[0].value =
➥marqueeString.substring(position++%sLength,position+39%sLength)
```

This line updates the value of the form element. The `substring()` method grabs a portion of the string in order to place it in the text input box. The `substring()` method also has two arguments: a beginning index and an end index. The beginning index is given as

```
position++%sLength
```

This statement indicates that after the index position is used, it should be updated by one (`position++`). The rest of the statement indicates that the actual index used is the modulus of the index with the string length. Using the modulus operator guarantees that position is always between zero and the length of the string. Figure 8.2 shows the marquee after it has begun scrolling out of the box.



Location: file:///Power%20Terminator/Desktop%20Folder/x

Welcome to my Web Page, I hope

Document: Done

Figure 8.2 *The marquee text form.*

The marquee form represents the diversity that can be achieved with the use of HTML forms. The idea that HTML forms should be used only for sending information *from* the user *to* the server ignores the potential of these objects. It is better to think of the form object as an interface between the user and your JavaScript programs that works in *both* directions.

In addition to JavaScript's capacity to take advantage of the updateability of the forms object, there is an additional feature that should be mentioned. Form elements retain their values in the Netscape browser between document loadings. If you have ever filled out a form, received a confirmation after submission, and then gone back to the form, you will notice that all your old entries are still there. This is called persistence and can be used to store information in a document when it might be left temporarily. There may occur a situation in which a form needs to be reloaded, or it might be left alone for a while, and you might want it to retain its final configuration when the user returns to it within the same session. If this is the case, you can create a text element or hidden form element and use it to store information. Whenever the page is loaded, you can check the contents of this field and rebuild the page according to the information in the form text or hidden element. Of course, if you want to retain information between sessions, when Netscape might be closed or your pages are removed from the cache, you will need to store this data in a "cookie," which is discussed in Chapter 7.

Elements of a Form

The test form given as an example at the beginning of the chapter has examples of all the form elements for which JavaScript has provided objects. These objects include the following:

- □ `buttons`
- □ `checkboxes`
- □ `passwords`
- □ `radios`
- □ `reset`
- □ `select`
- □ `submit`
- □ `text`
- □ `textarea`

The next few sections present the properties, methods, and attributes of these objects, as well as the form of their HTML tags.

To reference these elements in your JavaScript code, you must provide the array index for the specific element of the specific form in the document where the element exists, or use the element's name:

```
elementName.propertyName
elementName.methodName()
formName.elements[index].propertyName
formName.elements[index].methodName
```

If you know the element name beforehand, it is much easier and clearer to use the name itself instead of an index. At present, you cannot import JavaScript code using the SRC= attribute in the <SCRIPT> tag, so all your JavaScript code must necessarily exist on the same page as your HTML document. This means that using the actual names of form and form elements isn't a problem. However, when SRC attributes are allowed, the code you bring in could belong to a generic script, which can figure out the overall structure at runtime via the array indexing and length properties of arrays, without being hardwired for any particular HTML document. These types of JavaScript modules—which Web page developers merely call into their code to work—depend upon less intuitive array indexing.

The `button` Element

The `button` element provides a single event trigger that can be used for anything from submitting forms to calling up dialog boxes or changing the background color of the document. The HTML code for a `button` element is

```
<INPUT
    TYPE="button"
    NAME="button name"
    VALUE="button text"
    [onClick="JavaScript code"]>
```

The `button` object has the following members:

□ `name`. The `name` property is equivalent to the NAME attribute in the button HTML tag.

□ `value`. The `value` property is equivalent to VALUE attribute in the button HTML tag. The `value` property's text appears on the button when it is generated in the form.

□ `click()`. The `click()` method simulates the button being clicked; however, this does not call the `onClick` event handler. If you are trying to invoke the `onClick` action, you should call the JavaScript code in the handler directly. For example, if the `onClick` event handler were to call the `stop()` function instead of simulating the event by using *`buttonName`*`.click()`, you could simply place the `stop()` function in the handler.

□ `onClick`. The `onClick` event handler is called when the button is pressed by the user. The JavaScript code indicated in the string is executed by the interpreter.

The checkbox Element

The `checkbox` element provides a single box that can be either selected or unselected. You can think of this as being either true or false. If you have several check boxes, they can all be independently selected or deselected without affecting each other. This is different from the radio element, which ensures that only one element in the group is selected at a time. The HTML code for the check box is

```
<INPUT
    TYPE="checkbox"
    NAME="checkbox name"
    VALUE="checkbox value"
    [CHECKED]
    [onClick="JavaScript code"]>
text displayed next to checkbox
```

The `checkbox` object has the following members:

□ `checked`. This property reflects the present state of the box at runtime, indicating whether it is checked (true) or unchecked (false).

- □ defaultChecked. This property determines what the default or initial state of the check box is true for checked and false for unchecked.
- □ name. The name property is equivalent to the NAME attribute in the check box HTML tag.
- □ value. The value property is equivalent to the VALUE attribute in the check box HTML tag.
- □ click(). The click() method simulates the check box being selected, which toggles its state to the opposite of what it was before the click; however, this does not call the onClick event handler. If you are trying to invoke the onClick action, you should call the JavaScript code in the handler directly.
- □ onClick. The onClick event handler is called when the check box is selected by the user. The JavaScript code indicated in the string is executed by the interpreter.

The hidden Element

The hidden element is a text field that is not displayed in the onscreen document. This provides the JavaScript programmer with a persistent storage area in a document without resorting to the cookie property of a JavaScript document object. If you recall from the discussion earlier in the chapter, form elements retain their values between loading in a Navigator session. If you fill out a form document, leave the page and then return, the information in the form is retained. In addition to storing information that might be needed later when the user returns to the page in the same session, you might also want to submit information to the server that the user would not need to see in an ordinary form element (such as a regular text field). By using the hidden element, you can store the information and submit it just like any other form information. The hidden HTML tag is

```
<INPUT
   TYPE="hidden"
   NAME="hidden name"
   [VALUE="hidden text"]>
```

The hidden element does not have any methods or event handlers. However, it does have three properties:

- □ defaultValue. This property holds the default value that the hidden element stores if no other data is provided.
- □ name. This property is the NAME attribute of the hidden HTML tag.
- □ value. This property holds the value of the hidden text field and is equivalent to the VALUE attribute of the hidden HTML tag.

The password Element

The password element is similar to the text element, except it hides the actual text entered by replacing all the letters with an asterisk (*). This is the typical behavior for fields that accept passwords users would not want anyone looking over their shoulder to be able to see. The HTML tag for a password object is

```
<INPUT
    TYPE="password"
    NAME="password name"
    [VALUE="password text"]
    SIZE=integer
```

The password element has three properties, three methods, and no event handlers:

- □ defaultValue. The default value is similar to the defaultValue in the hidden object in that it provides the default string that the password object takes if none is provided.
- □ name. This property is the NAME attribute of the password HTML tag.
- □ value. This property holds the value of the password text field and is equivalent to the VALUE attribute of the password HTML tag.
- □ focus. This method causes the focus to shift to the password object so that it can accept user input.

□ blur. This method removes the input focus from the object. Input focus is required in order for the field to accept characters entered at the keyboard.

□ select. This method will cause the text in the password field to be highlighted.

The radio Element

The radio element is a radio button object that enables the user to select one button out of a group of objects at a time. This can be used for mutually exclusive items. To define a radio group, you provide them all with the same radio name:

```
<INPUT
    TYPE="radio"
    NAME="group name"
    VALUE="button value"
    [CHECKED]
    [onClick="JavaScript code"]>
    radio button text
```

Each individual radio object can be indexed in the elements array like any other form element; however, in addition to these methods, the radio group can be referenced by the group's name as an array.

```
groupName[index].property
```

The radio element has six properties, one method, and one event handler:

□ checked. This property holds the boolean value for the state of the radio button; true for checked, false for unchecked.

□ defaultChecked. This property indicates whether this is the default button to be checked out of the radio button group.

□ index. This property indicates which element this radio button is in the current group.

□ length. This property indicates the number of radio buttons in the group.

- □ name. This property holds the name of the radio element and is equivalent to the NAME attribute of the radio HTML tag.
- □ value. This property holds the value of the radio element and is equivalent to the VALUE attribute of the radio HTML tag.
- □ click(). The click() method simulates the radio button being selected, which will toggle its state to the opposite of what it was before the click; however, this does not call the onClick event handler. If you are trying to invoke the onClick action, you should call the JavaScript code in the handler directly.
- □ onClick. The onClick event handler is called when the radio button is selected by the user. The JavaScript code indicated in the string is executed by the interpreter.

The reset Element

The reset element provides a single trigger event that causes the form to clear all user entries, which represents it in its original state. Otherwise, it is much like the regular button element. The HTML code for a reset element is

```
<INPUT
    TYPE="reset"
    NAME="reset name"
    VALUE="reset text"
    [onClick="JavaScript code"]>
```

The reset object has the following members:

- □ name. The name property is equivalent to the NAME attribute in the reset HTML tag.
- □ value. The value property is equivalent to the VALUE attribute in the reset HTML tag.
- □ click(). The click() method simulates the button being clicked; however, this does not call the onClick event handler. If you are trying to invoke the onClick action, you should call the JavaScript code in the handler directly.
- □ onClick. The onClick event handler is called when the button is pressed by the user. The JavaScript code indicated in the string is executed by the interpreter.

The select Object

The select object is perhaps the most complex forms object you will have to deal with. It provides a means for creating a list of selections available to the user. These selections can appear in either scrolling or static windows, and the user can be restricted to the number of items that can be picked from the list. Because of this complexity, the HTML code is more involved. Additionally, the properties, methods, and event handlers are more complex as well. The HTML code for the selector is

```
<SELECT
    NAME="selector name"
    [SIZE="integer"]
    [MULTIPLE]
    [onBlur="JavaScript code"]
    [onChange="JavaScript code"]
    [onFocus="JavaScript code"]>
    <OPTION [SELECTED]> selection option
    [<OPTION> additional selections]
</SELECT>
```

The test form at the beginning of the chapter had a select form in it for choosing which computers the user owned:

```
<SELECT NAME="select1" MULTIPLE>
<OPTION>PC
<OPTION>UNIX
<OPTION>MAC
<OPTION>OTHER
</SELECT>
```

The MULTIPLE attribute controls the number of options available at one time. The SIZE attribute indicates how many options are visible at one time. The select object has nine properties, no methods, and three event handlers:

- □ length. This property is equivalent to the SIZE attribute in the select HTML tag.
- □ name. This property is equivalent to the NAME attribute in the select HTML tag.

- □ options. This is an array of the different options available to choose from. The array runs from options[0] for the first option to options[n-1] for the last option.

- □ selectIndex. This property indicates which item is selected in the select list. If the MULTIPLE attribute is set, then changing the selectedIndex value will clear all other selections since the index only refers to a single option.

  Of course, the options array is actually an array of objects, which have their own properties. Each of the individual options in a select element has the following properties:

  - □ defaultSelected. This is a boolean value indicating whether the option is automatically selected or unselected when it comes up.

  - □ index. This indicates where in the options list the current option is located.

  - □ selected. This indicates whether the option is currently selected and is equivalent to the HTML SELECTED tag.

  - □ text. This holds the text shown for each option.

  - □ value. This property holds the data sent to a CGI server from the select list if a submit button in the same form is pressed.

To access the individual option elements in a select element, you can either use the select object's name:

```
selectName.options[index].property
```

... or use the form name:

```
formName.elements[index1].options[index2].property
```

Notice that the select object has its own array in the document object, and can be accessed in a similar manner to forms.

The select object has three event handlers that enable the form to react to user input:

- □ onBlur. This event handler is called when the select element loses the focus. This typically occurs after the user has selected items in the list and then clicked another form element.
- □ onChange. This event handler is called when one of the options changes states in the select list. This can be from the user selecting or deselecting an item.
- □ onFocus. This event handler is called when the user clicks the select object and begins choosing items from the list.

The submit Element

The submit element provides a single trigger event that causes the form data to be sent to the URL specified in the HTML tag. Otherwise, it is much like the regular button element. The HTML code for a submit element is

```
<INPUT
    TYPE="submit"
    NAME="submit name"
    VALUE="submit text"
    [onClick="JavaScript code"]>
```

The submit object has the following members:

- □ name. The name property is equivalent to the NAME attribute in the submit HTML tag.
- □ value. The value property is equivalent to the VALUE attribute in the submit HTML tag.
- □ click(). The click() method simulates the button being clicked; however, this does not call the onClick event handler.
- □ onClick. The onClick event handler is called when the button is pressed by the user. The JavaScript code indicated in the string is executed by the interpreter.

The text Element

The text element enables the user to input short character sequences such as numbers, words, or a sentence. The text field was used in the marquee example to create a scrolling field in which the programmer provided the input to the field. The text object is defined in the HTML document as follows:

```
<INPUT
    TYPE="text"
    NAME="text name"
    VALUE="text value"
    SIZE=integer
    [onBlur="JavaScript code"]
    [onChange="JavaScript code"]
    [onFocus="JavaScript code"]
    [onSelect="JavaScript code"]>
```

In addition to the standard TYPE, NAME, and VALUE attributes, the SIZE attribute indicates how many characters the field can hold. The text element has three properties, three methods, and four event handlers.

- □ defaultValue. This property holds the default value string for the text element.
- □ name. This property is equivalent to the NAME attribute in the text HTML tag.
- □ value. This property is equivalent to the VALUE attribute in the text HTML tag.
- □ focus(). This method is used to move the input focus to this text element, causing the user's keyboard input to be directed to the text element.
- □ blur(). This method is used to cause the input focus to move away from the text element, causing the user's keyboard input to no longer be directed to the text element.
- □ select(). This method highlights the text in the text element.

- □ onBlur. This event handler is called when the text element loses the focus. This typically occurs after the user has typed text in the text element, and then clicked another element or pressed the Tab key.
- □ onChange. This event handler is called when the value of the text element changes. This is typically called when a user alters the entry in a text field. The event is called when the user leaves the field.
- □ onFocus. This event handler is called when the user clicks on the text object.
- □ onSelect. This event handler is called when the text in the text element is highlighted.

The textarea Element

The textarea element enables the user to input longer entries and is typically used for messages and comments that may span more than one line of text. The textarea object is defined in the HTML document as follows:

```
<TEXTAREA>
    NAME="textarea name"
    ROWS=integer
    COLS=integer
    [onBlur="JavaScript code"]
    [onChange="JavaScript code"]
    [onFocus="JavaScript code"]
    [onSelect="JavaScript code"]>
    Text displayed
</TEXTAREA>
```

Notice that instead of being a normal <INPUT> element, the textarea (like the select element) has its own name. In addition to the standard attributes, textarea also has ROWS and COLS. These attributes indicate how many lines deep, and how many characters wide the textarea should be, respectively.

- □ defaultValue. This property holds the default value string for the textarea element.
- □ name. This property is equivalent to the NAME attribute in the textarea HTML tag.

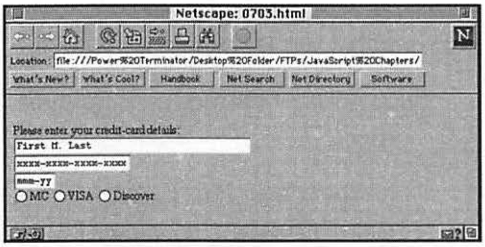
- □ `value`. This property is equivalent to the VALUE attribute in the textarea HTML tag.
- □ `focus()`. This method is used to move the input focus to the `textarea` element, causing the user's keyboard input to be directed to the `textarea` element.
- □ `blur()`. This method is used to cause the input focus to move away from the `textarea` element, causing the user's keyboard input to no longer be directed to the `textarea` element.
- □ `select()`. This method highlights the text in the `textarea` element.
- □ `onBlur`. This event handler is called when the `textarea` element loses the focus. This typically occurs after the user has selected items in the list, and then clicked another element.
- □ `onChange`. This event handler is called when the value of the `textarea` element changes states. This is typically called when a user alters the entry in a `textarea` field. The event is called when the user leaves the field.
- □ `onFocus`. This event handler is called when the user clicks on the `textarea` object.
- □ `onSelect`. This event handler is called when the text in the `textarea` element is highlighted.

Event Handlers in Forms Object Elements

Use of the event handlers in the forms object elements is perhaps the most utilitarian capability JavaScript maintains. You have already seen the event handlers for forms used in many of the examples, in addition to event handlers for other objects. However, as a final example of the forms object in action, it is important that you see how a JavaScript program might verify user input before it is sent to the server.

Using JavaScript to Verify Forms

In order to verify entries in a form, it is necessary to check to see what the user has entered before it is sent to the server. To do this with `text` elements, anytime the user changes the text in the field, you can check to see whether it satisfies the requirements for the entry. Take the form in Figure 8.3, for example:



Location: file:///Power%20Terminator/Desktop%20Folder/FTPs/JavaScript%20Chapters/

Please enter your credit-card details:

First M. Last

xxxx-xxxx-xxxx-xxxx

mmm-yy

○ MC ○ VISA ○ Discover

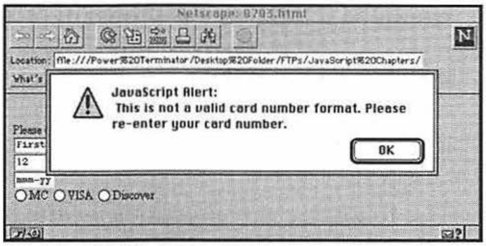
Figure 8.3 *A sample credit card form.*

If you wanted to make sure that everyone filled in the credit card number using the correct four-digit sequences with dashes in between, you could create the following form and script:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
function checkNumber(number) {
    if(number.indexOf("-") != 4 ¦¦ number.indexOf("-",5) != 9 ¦¦
➥number.indexOf("-",10) != 14) {
        alert("This is not a valid card number format. Please re-
➥enter your card number.")
        return false
    }
}

// -->
</SCRIPT>
</HEAD>
<BODY>
<PRE>
</PRE>
<FORM NAME="data">
Please enter your credit-card details:<BR>
<INPUT TYPE="text" NAME="Name" VALUE="First M. Last" SIZE=40><BR>
```

```
<INPUT TYPE="text" NAME="cardNumber" VALUE="xxxx-xxxx-xxxx-xxxx"
➥SIZE=19 onChange="checkNumber(this.value)"><BR>
<INPUT TYPE="text" NAME="expires" VALUE="mmm-yy" SIZE=6><BR>
<INPUT TYPE="radio" NAME="cardType" VALUE="MC">MC
<INPUT TYPE="radio" NAME="cardType" VALUE="VA">VISA
<INPUT TYPE="radio" NAME="cardType" VALUE="DR">Discover
</FORM>
</BODY>
</HTML>
```

The `onChange` event handler for the `cardNumber` element causes the `checkNumber` function to be called. This results in the number being checked for the correct dashes. If a user attempts to leave the field with an incorrect entry, he or she receives the warning shown in Figure 8.4.


Netscape: 0703.html
Location: file:///Power%20Terminator/Desktop%20Folder/FTPs/JavaScript%20Chapters/
What's
Please
First
12
mmm-yy
○MC ○VISA ○Discover
OK

JavaScript Alert:
This is not a valid card number format. Please re-enter your card number.

Figure 8.4 *The Alert dialog box for form verification.*

The possibilities for what you can do with the form elements in a document are almost unlimited. This is one of JavaScript's strong points. Be sure to take the time to try out several of the event handlers and form element properties.

Controlling Window Objects

At the top of the Navigator object hierarchy is the window object. This object represents the browser window that exists on the client machine, and can be used to control the screen real estate for your documents. The window object consists of several properties, methods, and event handlers that can be used to control various aspects of how the browser window is displayed. The window object also controls dialog boxes, frames, and timeouts. You have already seen or used many of these in example programs before.

There are two HTML tags that correspond to elements of the Netscape window object. These are the <BODY></BODY> and <FRAMESET></FRAMESET> tags. In particular, in the HTML body tag, the `onLoad` and `onUnLoad` event handlers are properties of the window object. The frameset tag is reflected in the `frames` property array that holds an element for every frame in the window. The `onLoad` event handler was used for the marquee example in the beginning of the chapter, and the `frames` property was used in the directory example from Chapter 6, "JavaScript and Built-In Objects."

The window object does not need to be directly referenced in a JavaScript call, so using `window` in

```
window.alert("don't do that!")
```

would be redundant; as shown in previous examples, only

```
alert("don't do that!")
```

is necessary.

The following three sections list the properties, methods, and event handlers for the window object, and provide reference to examples that have used them.

Properties of the Window Object

With Netscape Navigator 2.0, the possibility exists to have multiple targets for an HTML document's links or function calls that can either be additional windows or frames within windows. For the most part, the properties of the window object hold references to these additional windows and frames that may be created in the Navigator environment.

- □ `defaultStatus`. This property is a string that holds the default message displayed in the status bar of the window being referenced. In many cases you will want to change the message in the status bar during an `onMouseOver` event, where it changes depending upon what object the mouse is over at the time. In order to do this, you must make sure the event handler returns true in order for the status to be updated.

□ frames. The frames property is an array that holds all of the frames of the window. These are indexed from 0 to the number of frames. The frames property refers to the frames in the order they are declared in a <FRAMESET> HTML tag. You can then direct your method calls to the particular frame you need. The frame object itself is a kind of window, inheriting all the properties and methods that the window object has, so referencing window.frames[0].defaultStatus refers to the default status text displayed when the mouse is in the first frame of the window.

□ parent. This refers to the window in which the frame(s) being referenced reside. If you recall from the directory example at the end of Chapter 5, in order to have a document in a frame refer to a function in the overall window, it has to be prefixed with a parent. The following portion of the code from that example calls the overall window's function displayRoot() from within a frame. This code will not work by itself because it's part of the larger example in Chapter 5:

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<SCRIPT LANGUAGE="JavaScript">
<!--
parent.displayRoot()
// -->
</SCRIPT>
</BODY>
</HTML>
```

□ self. This refers to the current window. This is not normally needed; however, sometimes it makes code easier to read if you refer to the current window as self.

□ status. This property holds the current text in the status frame of the document window. You can set this status text to prompt the user when they move the mouse over a link or object that uses the onMouseOver event handler. Remember, you must return true from the onMouseOver event handler to set the status text correctly:

```
<A HREF="http://home.netscape.com/" onMouseOver="self.status='Home
of JavaScript!'; return true">
```

□ `top`. This property holds the window used to create all subwindows. If you notice in the File menu item in Netscape, you can open a new browser window. You can do the same thing using the `window.open()` method, which will be covered in the next section. This was the function used to create the flashing billboard from Chapter 7. When you create new windows like this and open documents in them, the documents can refer to the topmost window used to create any child windows such as the one the flashing billboard was displayed in.

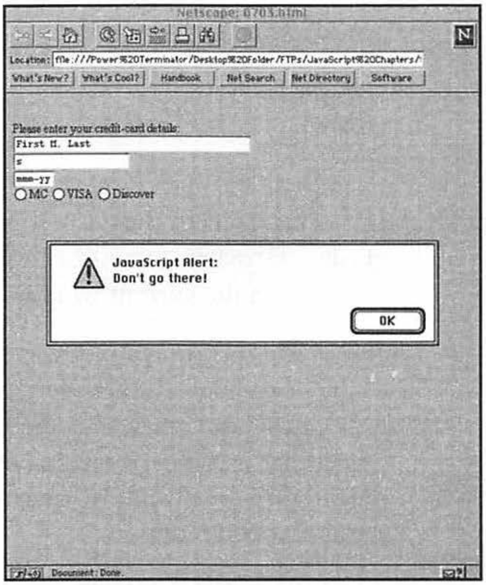
□ `window`. This property refers to the current window and is identical in use to the `self` property.

Methods of the Window Object

□ `alert()`. The `alert()` method creates a dialog box with a single OK button. In the dialog itself, the programmer can specify what he or she wants the alert to say.

```
alert("Don't go there!")
```

This method has been used several times in previous code.



Netscape: 0703.html
Location: file:///Power%20Terminator/Desktop%20Folder/FTPs/JavaScript%20Chapters/
What's New?
What's Cool?
Handbook
Net Search
Net Directory
Software

Please enter your credit-card details:

First M. Last

mmm-yy

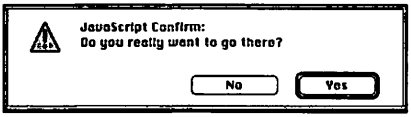
○ MC ○ VISA ○ Discover

JavaScript Alert:
Don't go there!

Figure 8.5 *The Alert dialog box.*

□ close(). The close() method is used to close the window and should not be confused with the document.close() method, which closes the document stream from the host server.

□ confirm(). The confirm() method is similar to the alert() in that it creates a dialog box with the text that the user provides. However, this dialog box has two buttons: OK and Cancel. If the user selects OK, then the method returns true; if the user selects Cancel, then the method returns false.



JavaScript Confirm:
Do you really want to go there?

Figure 8.6 *The confirm dialog box.*

□ open(). The open() method, not to be confused with the document.open() method, is used to open a new window. The default arguments that the method takes are

```
window.open("URL", "window name", ["window features"])
```

The URL and window name provide the document and name of the new window to be opened. The window features argument is a string that holds the number of arguments separated by commas with no spaces about how the window is to be opened. In the case of the document example from Chapter 6, the method call was

```
msgWindow = window.open("","Hello",
"toolbar=no,width=200,height=100")
```

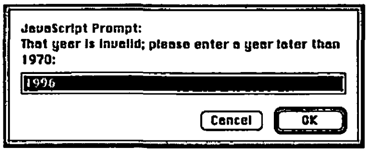
Notice the use of the "toolbar=no,width=200,height=100" string to set the parameters of the window. These parameters can only be set for new windows being opened, and will not affect the parent or top window. The full list of valid property arguments are

```
toolbar=yes|no
location=yes|no
directories=yes|no
status=yes|no
```

```
menubar=yes¦no (Windows only)
scrollbars=yes¦no
resizable=yes¦no (Windows only)
width=pixels
height=pixels
```

- □ `prompt()`. The `prompt()` method provides an alternative to the confirm dialog box by enabling the user to input a replacement value in the case of an erroneous or invalid entry. In order to call the prompt dialog box, you must provide the message and an optional default value for the input field:

```
prompt("That year is invalid; please enter a year later than
1970:", 1996)
```



JavaScript Prompt:
That year is invalid; please enter a year later than 1970:

1996

Figure 8.7 *The prompt dialog box.*

- □ `setTimeout()`.
- □ `clearTimeout()`. These two methods provide a means for calling a JavaScript statement after a certain time period has elapsed. If you recall the marquee example from the beginning of the chapter, the `timeout` methods were used to control the updating of the text element.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--

var buffer = "                     "
var text = "Welcome to my Web Page, I hope you enjoy yourself!"
var marqueeString = buffer + text + buffer
var marqueeRunning= false
var timeoutID = null
var position = 0
var sLength = marqueeString.length
```

```
function stopMarquee() {
    if(marqueeRunning)
        clearTimeout(timeoutID)
    marqueeRunning = false
}

function startMarquee() {
    stopMarquee()
    updateMarquee()
}

function updateMarquee() {
        document.marquee.elements[0].value =
➥marqueeString.substring(position++%sLength,position+39%sLength)
        timeoutID = setTimeout("updateMarquee()", 50)
        marqueeRunning = true
}

// -->
</SCRIPT>
</HEAD>
<BODY onLoad="startMarquee()" onUnLoad="stopMarquee()">
<PRE>
</PRE>
<FORM NAME="marquee">
<INPUT TYPE="text" NAME="textBox" VALUE="" SIZE=40><BR>
</FORM>
</BODY>
</HTML>
```

Event Handlers in the Window Object

The window object contains two event handlers:

- □ onLoad
- □ onUnLoad

These event handlers were used in the marquee example to start and stop the updating of the text element when the document was loaded and unloaded, respectively.

Using the History and Location Objects

The history and location objects provide a means for accessing URL links that the user has recently been to or is currently viewing. This gives the JavaScript programmer the ability to make his or her script "environmentally aware." This means the scripts can tell where their users are (that is, the location object) and where their users have been (that is, the history object). By using these objects you can create a powerful tool for navigating through your site based upon where a user is and where he or she has been.

The History Object

The history object provides the JavaScript programmer with the ability to select and load URLs from the Go menu. This ability can provide you with your own forward and back controls for documents in your site, based upon where the user has actually been.

Of course, the ability to access the link history should be used carefully, especially when you begin to look at where people have been besides your own server. Because it is possible to load information from the history list into hidden text forms and send them to a server, it is possible to mine information from a user's browser about how he or she reached your site. However, if you secretly download information about places that the user wouldn't necessarily want people to know he or she had been, then you might start to raise a few ethical flags in the minds of your site's users. If you decide to use the history information for anything more than allowing them to skip back and forth easily within your own site, be sure to get permission first. Have an introduction document or "cover page" for your site that tells people that their link history will be viewed. This admonition also goes for downloading *any* type of information from a client machine that JavaScript may expose.

Properties in the History Object

The only property in the history object is `length`. This provides the number of links in the history object. Remember, the history list goes both backward and forward.

Methods in the History Object

There are three methods in the history object:

- □ `back()`. The result of calling the `history.back()` method is identical to the user selecting the back arrow on the toolbar in Netscape.
- □ `forward()`. The result of calling the `history.forward()` method is identical to the user selecting the forward arrow on the toolbar in Netscape.
- □ `go(int)`. The `go()` method takes an integer (positive or negative) and loads the link that is that number of places forward or backward in the history list. For example, `history.go(-3)` moves the user to the link he or she was at three clicks ago.

The Location Object

The location object is used to store information about the current URL. You can use the property names to get to the relevant information in the URL that you need such as the host, protocol, or pathname. The location object is a static object. It stores the relevant information from the current URL. If you want to change the location information, you must use this object; do not confuse the location object with the document.location property. While the document.location property is often the same as the location object's URL, it can be changed by code. Remember, the document.location *property* is read-only.

URLs can be divided into several sections. For example, take the following URL:

```
http://www.mcp.com:80/hayden/index.html#start
```

The URL can be parsed as follows. The http section of the URL is called to protocol. Other protocols include mailto and ftp. The www.mcp.com represents the hostname. This can also be a series of numbers indicating the IP address of the server. 80 is the port number that the server is listening to. /hayden/index.html is the pathname, while #start is the hash. In addition to the hash, which refers to an anchor in a document, you can also have ?query=no, which would be a search for a server-side CGI program.

All these different values are stored in the location properties.

□ `hash`.

□ `host`. This is a combination of `hostname:port`.

□ `hostname`.

□ `href`. This represents the entire URL.

□ `pathname`.

□ `port`.

□ `protocol`.

□ `search`.

Moving on to Java

Up to now, this book has covered the use of the JavaScript language to create dynamic Web sites and control the display of Web documents. While JavaScript provides a powerful set of tools for Internet development, there is still a key limitation to its approach—you are still using the same basic elements that all Web pages can have, except now you can provide them with the capability to respond to user-generated events. Java, on the other hand, provides a completely new framework upon which you can build innovative and powerful content that is not available in the standard HTML toolkit. In fact, Java enables you to create your *own* tools and incorporate new tools that others have created.

If you plan to learn Java in conjunction with or after mastering JavaScript, here are some of the key concepts you'll want to keep in mind as you make the transition to this considerably more complex, yet feature-rich, programming environment.

□ Java is truly object-oriented. Remember those terms we chose not to worry about for JavaScript, like class inheritance and encapsulation? Well, they're back with a vengeance in Java. Everything you do in Java will require you to think of your programs, functions, interfaces, and even numbers and characters as objects. For example, objects of one kind cannot access the methods of other objects or mess with their settings and properties; they're all self-contained. If you already have a mindset in which your programs are composed of a number of functions and some global variables that get changed by these functions as the program runs step-by-step, you'll need to rearrange your brain a bit. Objects don't work like conventional programming languages' procedures and subroutines do, so a lot of your BASIC or PASCAL knowledge will not readily transfer to an object-oriented language such as Java. We'd like to suggest *Teach Yourself Java for Macintosh in 21 Days* (Hayden Books) as a reference to take you from where you are now to proficiency with objects and Java.

□ Java can create stand-alone applications and Web-accessible "applets." Because you now know that Java is a complete programming environment, you should understand exactly what you can create with it. Because your interest in the World Wide Web has led you to learn Java, you have probably heard of Java "applets" becoming the Next Big Thing. Java programmers can create applications that are meant to be run within a Web page and interact with their end user through a Web browser's window. These applications are *applets,* and as a Macintosh enthusiast you'll be pleased to know that the latest Netscape Navigator versions support applet-running on Macs.

What you might not know is that Java can create stand-alone applications, or programs that do not run with Netscape or any other browser environment. These stand-alone applications run autonomously in much the same way as the Calculator desk accessory works all by itself as soon as you choose its icon in the Apple menu. As of this writing, the Macintosh Java Developer's Kit, available from Sun at http://www.javasoft.com, does not support stand-alone application development for the Macintosh. (However, Natural Intelligence's Roaster, among others, are products that enable you to develop both stand-alone programs *and* applets.) While you may plan to create Java programs exclusively for Web use as applets, understanding Java from both the applet and stand-alone perspectives improves your ability to understand potential bugs in your own code and reduces some of the mystery of why Java works the way it does.

- □ Java is *not* coded like JavaScript. Recall the original "spirit" and purpose of JavaScript: to enable nonprogrammers to be able to improve the interactivity of their own Web sites. Thus, JavaScript is coded right into the HTML tags of your existing Web pages and uses a greatly simplified programming syntax when compared to Java. Java applets and stand-alone programs must be developed in their own separate development environment and use a richer, stricter syntax. Programmers who have experience with C or C++ will be much more comfortable than those with experience in BASIC or other less rigorous languages.

Don't let all this gloom and doom about "complexity" and "rigor" bother you. What you've already learned about JavaScript objects and syntax should help you learn Java much more quickly than if you started Java cold. (There's a coffee pun in there somewhere.) If you're ready to accept the challenge, dive in head-first. The beauty of software experimentation is that you can't break or use up any physical resources while learning (unless you let your temper get the better of you). It's not like you end up having to buy a pricey replacement block of mahogany because you got a little lathe-happy as the carpenter's apprentice.

Summary

This chapter provided you with a look at almost all the remaining JavaScript objects you can use in your HTML documents. You started out in this book by learning the fundamentals of the JavaScript language and how to place scripts within HTML documents. The chapters then moved on to cover the basic foundations of the language, and on to more complicated topics such as functions and objects. Once past that stage, the last few chapters have given you a guided tour of the different JavaScript objects you can employ in your Web documents, with examples to show you what is possible. At this point, you are ready to dive into JavaScript on your own (assuming you've shown super-human willpower and haven't hacked around on your own already). Take the examples and try altering them, and certainly try creating your own, based on the ideas that motivated you to read this book in the first place. Practice, creativity, and knowledge of syntax help make efficient, effective programmers. A word of caution: JavaScript still has a few minor bugs that did not work themselves out in the first release, and it is far from a complete package. Several of the advertised features, such as the capability to interact with Java applets and the navigator plug-ins, have not materialized in the 2.0 release, so don't be surprised if something doesn't work like you expect (or hope).

Now that you have completed this book, you should consider it a complete reference for JavaScript instead of primarily a how-to guide. You will no doubt forget syntax from time to time, especially since JavaScript uses a number of unintuitive characters like ¦ and ! as logical or mathematical operators, for example. As you build your cool new JavaScript pages, remember to keep your users in mind. Show your page prototype to a friend or someone you know on the Net. Ask them for feedback, especially if your page is highly interactive or requests complex information from its users to work its magic. Make changes based on their feedback. Many of the "worst" programs start as fantastic functional ideas that unfortunately reflect no one's way of thinking about the problem except the designer's. And fool ourselves as we may, ***we are not our own Web sites' users.***

APPENDIX A

Object Reference

This appendix lists all of JavaScript's objects and each object's applicable methods, properties, and event handlers. Additionally, each object includes a syntax definition.

Objects with the (Client) notation are implemented by Web browsers that support JavaScript, such as Netscape Navigator 2.0J. Objects with the (Common) notation are *static* objects, which are not dependent upon a particular browser for correct implementation in JavaScript code.

Anchor (Client)

HTML anchors enable you to create hyperlink reference points within a document. An anchor tag names a specific index point within a document for reference via links.

JavaScript can access and take advantage of the various anchors within a document.

HTML Syntax:

```
<A Name="anchorName">anchor text</A>
```

Button (Client)

Executes an action or a JavaScript script.

HTML Syntax:

```
<INPUT TYPE="Button" NAME="objectName" VALUE="buttonText"
➥[onClick="handlerText"]>
```

INPUT TYPE: The type of button or the generic "button"

NAME: A name for the button object

VALUE: Any text that is to appear on the button

onClick option: A JavaScript function

Properties:

□ name

□ value

Methods:

□ click

Event Handlers:

□ onClick

Checkbox (Client)

Creates and identifies a checkbox object on any HTML form.

HTML Syntax:

```
<INPUT TYPE="Checkbox" NAME="checkboxName" VALUE="Checked"
➥[Onclick="handler"]> Checkbox text
```

Properties:

□ `checked`

□ `defaultChecked`

□ `name`

□ `value`

Methods:

□ `click`

Event Handlers:

□ `onClick`

Date (Common)

The date object enables the manipulation of dates and times within HTML documents.

Syntax:

```
objectname = new Date(Parameters)
```

Where `Parameters` =

□ A string of form "Month Day, Year Hours:Minutes:Seconds"

□ An Integer of Form Year, Month, Day, Hours, Minutes, Seconds

Methods:

□ `getDate`

□ `getDay`

□ `getHours`

□ `getMinutes`

- □ `getMonth`
- □ `getSeconds`
- □ `getTime`
- □ `getTimeZoneoffset`
- □ `getYear`
- □ `parse`
- □ `setDate`
- □ `setHours`
- □ `setMinutes`
- □ `setMonth`
- □ `setSeconds`
- □ `setYear`
- □ `toString`
- □ `toGMTString`
- □ `toLocaleString`
- □ `UTC`

Document (Client)

Text information in the current HTML document.

HTML Syntax: The document object encloses the text within the <BODY> </BODY> HTML tags.

Properties:

- □ `alinkColor`
- □ `anchors`

- □ `bgColor`
- □ `cookie`
- □ `fgColor`
- □ `forms`
- □ `lastModified`
- □ `linkColor`
- □ `links`
- □ `location`
- □ `referrer`
- □ `title`
- □ `vlinkColor`

Methods:

- □ `clear`
- □ `close`
- □ `open`
- □ `write`
- □ `writeln`

Form (Client)

Forms enable client data retrieval and user data entry and act as a document's interface to a remote server for CGI scripts, or for JavaScript user input.

HTML Syntax:

```
<FORM NAME="name" Target="windowName" ACTION="Relative Path or URL"
METHOD=GET "Post1" [onSubmit="handler"]></Form>
```

Properties:

- □ `action`
- □ `elements`
- □ `encoding`
- □ `method`
- □ `target`

Methods:

- □ `submit`

Event Handlers:

- □ `onSubmit`

Frame (Client)

A standard HTML frame (a Netscape proprietary tag as of this writing), commonly defined as a separate region in an HTML document.

HTML Syntax:

```
<FRAMESET> ... </FRAMESET>
```

Properties:

- □ `defaultStatus`
- □ `frames`
- □ `parent`
- □ `self`

□ `status`

□ `top`

□ `window`

Methods:

□ `alert`

□ `close`

□ `confirm`

□ `open`

□ `prompt`

□ `setTimeout`

□ `clearTimeout`

Hidden (Client)

HTML text that is hidden from the client's view. Used with forms to store information.

HTML Syntax:

```
<INPUT TYPE= "hidden" NAME="textname" VALUE="textvalue">
```

Properties:

□ `defaultValue`

□ `name`

□ `value`

History (Client)

Browser's current list of sites its user has visited during the current usage session.

Syntax: `history.property` or `history.method`

Properties:

□ `length`

Methods:

□ `back`

□ `forward`

□ `go`

Link (Client)

HTML text identified as a hypertext link to another HTML document location.

Properties:

□ `target`

Event Handlers:

□ `onClick`

□ `onMouseOver`

Location (Client)

Syntax: `location.property` or `location.method`

Properties:

□ `hash`

□ `host`

□ `hostname`

□ `href`

□ `pathname`

□ `port`

□ `protocol`

□ `search`

Math (Common)

Built-in mathematical functions for use within HTML documents.

Syntax: `Math.property` or `Math.method`

Properties:

□ `E`

□ `LN10`

□ `LN2`

□ `PI`

□ `SQRT1_2`

□ `SQRT2`

Methods:

- □ `abs`
- □ `acos`
- □ `asin`
- □ `atan`
- □ `ceil`
- □ `cos`
- □ `exp`
- □ `floor`
- □ `log`
- □ `max`
- □ `min`
- □ `pow`
- □ `random`
- □ `round`
- □ `sin`
- □ `sqrt`
- □ `tan`

Navigator (Client)

Identifies the Netscape Navigator client being used to view the current HTML document.

Syntax: `navigator.property`

Properties:

- □ appName
- □ appVersion
- □ appCodeName
- □ userAgent

Password (Client)

A form object with special hidden text properties that replace standard keystroke characters with asterisks.

HTML Syntax:

```
<INPUT TYPE = "password" NAME="passwordname" VALUE="textvalue"
➥SIZE=integer>
```

Properties:

- □ password
- □ defaultValue
- □ name
- □ value

Methods:

- □ focus
- □ blur
- □ select

Radio (Client)

Radio buttons within an HTML form.

Properties:

- □ `checked`
- □ `defaultChecked`
- □ `index`
- □ `length`
- □ `name`
- □ `value`

Methods:

- □ `click`

Event Handlers:

- □ `onClick`

Reset (Client)

Identifies the RESET button on an HTML form.

HTML Syntax:

```
<INPUT TYPE="reset" NAME="resetbuttonname" VALUE="buttontext"
➥onClick="handler">
```

Properties:

- □ `name`
- □ `value`

Methods:

- □ `click`

Event Handlers:

- □ `onClick`

Select (Client)

A scrolling or standard selection list within an HTML form.

Properties (select):

- □ `length`
- □ `name`
- □ `options`
- □ `selectedIndex`

Properties (options):

- □ `defaultSelected`
- □ `index`
- □ `selected`
- □ `text`
- □ `value`

Event Handlers:

- □ `onBlur`
- □ `onChange`
- □ `onFocus`

String (Common)

A standard string of characters. A string can be manipulated using any of the following listed methods.

Syntax: `stringname.property` or `stringname.method`

Properties:

- □ `length`

Methods:

- □ `anchor`
- □ `big`
- □ `blink`
- □ `bold`
- □ `charAT`
- □ `fixed`
- □ `fontcolor`
- □ `fontsize`
- □ `indexOf`
- □ `italics`
- □ `lastIndexOf`
- □ `link`
- □ `small`
- □ `strike`
- □ `sub`
- □ `substring`

□ `sup`

□ `toLowerCase`

□ `toUpperCase`

Submit (Client)

SUBMIT button on an HTML form that sends form data to a remote server (CGI) or to JavaScript functions for processing.

Properties:

□ `value`

Methods:

□ `click`

Event Handlers:

□ `onClick`

Text (Client)

Any forms-based HTML entry field on the current HTML form.

Properties:

□ `defaultValue`

□ `name`

□ `value`

Methods:

□ `focus`

□ `blur`

□ `select`

Event Handlers:

- □ `onBlur`
- □ `onChange`
- □ `onFocus`
- □ `onSelect`

Textarea (Client)

Similar to the text object, this object identifies any multiline text entry field within the current HTML form.

Properties:

- □ `defaultValue`
- □ `name`
- □ `value`

Methods:

- □ `focus`
- □ `blur`
- □ `select`

Event Handlers:

- □ `onBlur`
- □ `onChange`
- □ `onFocus`
- □ `onSelect`

Window (Client)

The highest object level within the Netscape Navigator client. Contains document and other subobjects.

Properties:

- □ `defaultStatus`
- □ `frames`
- □ `parent`
- □ `self`
- □ `status`
- □ `top`

Methods:

- □ `alert`
- □ `close`
- □ `confirm`
- □ `open`
- □ `prompt`
- □ `setTimeout`
- □ `clearTimeout`

Event Handlers:

- □ `onLoad`
- □ `onUnload`

APPENDIX B

JavaScripts from Around the Web

We find that one of the best ways to truly learn JavaScript is to see it in action, then see the code that makes it work. That's what this appendix is all about. We've sought out some cool things that people are currently doing with JavaScript, and they've generously allowed us to include their code here, so you can dig through it and see just how they work their magic.

NOTE We've also included the code on the CD-ROM, so you can play with it a bit. Keep in mind, however, that the code is the property of its creators; make sure to contact them before using it or altering it in any way. Also note that we've put just the code on the CD, not all the graphics—all we care about is the functionality anyway, right?

For each example, we've included the URL of the page, info about its creator, any special uses of JavaScript to watch for, a screenshot, and of course the complete code for the page. And now, on with the show! Let's see what types of things the JavaScript pioneers are up to.

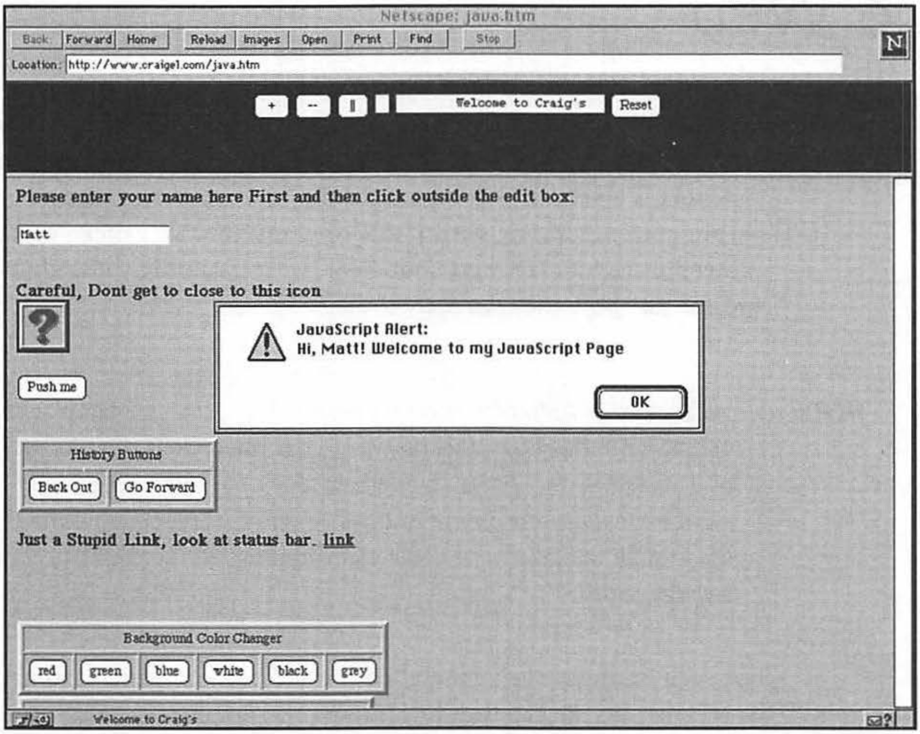
And remember: The Web is populated with a large number of innovative individuals (like yourself) who want to test the limits of JavaScript and their own creativity. Check your favorite search engines often for new Web sites that explore interesting JavaScript implementations. In case you don't already have a favorite search engine to visit, try Excite http://www.excite.com, Yahoo http://www.yahoo.com, or InfoSeek http://www.infoseek.com.

Craig's JavaScript Page

http://www.craigel.com/java.htm

Craig Slagel, craigel@nando.net

All sorts of goodies: scrolling text with controls, some rollover, and even buttons to change the background color.



Welcome to Craig's

Please enter your name here First and then click outside the edit box:

Matt

Careful, Dont get to close to this icon

JavaScript Alert:
Hi, Matt! Welcome to my JavaScript Page

Just a Stupid Link, look at status bar. link

Welcome to Craig's

```
<HTML>
<head>

<FRAMESET COLS="100%">

  <FRAMESET ROWS="15%,85%">
    <FRAME SRC="banner.htm" SCROLLING=NO>
    <FRAME SRC="javatest.htm" NAME="main">
  </FRAMESET>
```

```
</FRAMESET>

<NOFRAMES>

<P> You do not have a browser that can view frames. To view this page first go to and get <A
HREF="http://home.netscape.com/comprod/mirror/index.html">Netscape 2.0 beta</a> <A
HREF="http://home.netscape.com/comprod/mirror/index.html"><IMG SRC="now8.gif" VSPACE=1
WIDTH=88 HEIGHT=31 BORDER=1 Alt="Netscape now"></A>

</head>
</HTML>
```

banner.htm

```
<html>
<head>

<script language="javascript">

tid = 0;
pause = 0;
var to;
var bcount;
var tcount;

function bannerArray() {

   this.length = 54;

   this[1] = "                           W";
   this[2] = "                          We";
   this[3] = "                         Wel";
   this[4] = "                        Welc";
   this[5] = "                       Welco";
   this[6] = "                      Welcom";
   this[7] = "                     Welcome";
   this[8] = "                    Welcome ";
   this[9] = "                   Welcome t";
   this[10] = "                 Welcome to";
   this[11] = "                Welcome to ";
   this[12] = "               Welcome to C";
   this[13] = "              Welcome to Cr";
   this[14] = "             Welcome to Cra";
   this[15] = "            Welcome to Crai";
   this[16] = "           Welcome to Craig";
   this[17] = "          Welcome to Craig'";
   this[18] = "         Welcome to Craig's";
   this[19] = "        Welcome to Craig's ";
   this[20] = "       Welcome to Craig's P";
   this[21] = "      Welcome to Craig's Pa";
   this[22] = "     Welcome to Craig's Pag";
   this[23] = "    Welcome to Craig's Page";
   this[24] = "   Welcome to Craig's Page ";
   this[25] = "  Welcome to Craig's Page  ";
   this[26] = " ";
   this[27] = "  Welcome to Craig's Page  ";
   this[28] = " ";
   this[29] = "  Welcome to Craig's Page  ";
   this[30] = " Welcome to Craig's Page   ";
   this[31] = "Welcome to Craig's Page    ";
   this[32] = "elcome to Craig's Page     ";
   this[33] = "lcome to Craig's Page      ";
   this[34] = "come to Craig's Page       ";
   this[35] = "ome to Craig's Page        ";
   this[36] = "me to Craig's Page         ";
   this[37] = "e to Craig's Page          ";
   this[38] = " to Craig's Page           ";
   this[39] = "to Craig's Page            ";
   this[40] = "o Craig's Page             ";
   this[41] = " Craig's Page              ";
   this[42] = "Craig's Page               ";
   this[43] = "raig's Page                ";
   this[44] = "aig's Page                 ";
   this[45] = "ig's Page                  ";
   this[46] = "g's Page                   ";
   this[47] = "'s Page                    ";
   this[48] = "s Page                     ";
   this[49] = " Page                      ";
   this[50] = "Page                       ";
   this[51] = "age                        ";
   this[52] = "ge                         ";
   this[53] = "e                          ";
   this[54] = "                           ";

   return this
}

function MakeArray(n) {
this.length = n;
return this
}

banner = new bannerArray();
bannerstep = 1;
```

```
function banner1(n) {
              tid=window.setTimeout('banner1(bannerstep)",to);
              f.result.value = banner[bannerstep];
                window.status = banner[bannerstep];
              bannerstep = bannerstep + 1;
              if (bannerstep == 55) {
                     bannerstep = 1;
                            window.clearTimeout(tid);
                            tid=window.setTimeout("banner1()',to);
                     }
              }

function start(x) {
              f=x;
                to=60;
              banner1(x);
              }

function pausing(x) {
       if (pause == 0) {
              pause = 1;
              f.P.value = "P";
              window.clearTimeout(tid);
       }
       else {
              pause = 0;
              f.P.value = " ";
              banner1(x);
       }
}

function speedup() {
       if (to != 0) {
              to = to-30;
              }
}

function slowdown() {
       to = to+30;
}

function refr() {
       to = 90;
       bannerstep = 1;
}
```

```
function cleartids() {
       window.clearTimeout(tid);
}

</script>

<TITLE>Banner</TITLE>

</head>

<body bgcolor="#000000" text="#ff00ff" onload="start(document.forms[0])" onunload="cleartids()" >

<center>
<form name="banner">
<input type="button" name="+" value=" + " onClick="speedup()">
<input type="button" name="-" value=" -- " onClick="slowdown()">
<input type="button" name="pause" value=" ¦¦ " onclick="pausing(this.form)">
<input type="text" name="P" size=1>
<input type="text" name="result" size=28>
<input type="reset" name="restart" value="Reset" onclick="refr()">
</center>
</form>

<!-- Code was developed thanks to the code at http://www-ece.rice.edu/~vijaypai/chant.html
Code Modified by Craig Slagel http://www.boots.com/~craigel/craig.htm
-->

</html>
```

javatest.htm

```
<html>
<head>

<script language="LiveScript">

<!-- Hiding

     function hello() {
       alert("Hello! Did I make you Jump");
     }

     function WinOpen() {
        msg=open("","DisplayWindow","toolbar=yes,directories=no,menubar=no");
        msg.document.write("<HEAD><TITLE>Yo!</TITLE></HEAD>");
        msg.document.write("<CENTER><h1><B>A new window thanks to javascript!</B></h1></CENTER>");
     }
```

```
 function getname(str) {
    alert("Hi, "+ str+"! Welcome to my JavaScript Page");
  }

function changeBackground(hexNumber)
{
        document.bgColor=hexNumber
}

prefix="#"
rnum1=0
bnum1=0
gnum1=0
rnum2=0
bnum2=0
gnum2=0
hexNumber2="#000000";
rcount=0;
bcount=0;
gcount=0;

function num2hex(num) {
       if (num==15) return "f";
       else if (num==14) return "e";
       else if (num==13) return "d";
       else if (num==12) return "c";
       else if (num==11) return "b";
       else if (num==10) return "a";
       else if (num==9) return "9";
       else if (num==8) return "8";
       else if (num==7) return "7";
       else if (num==6) return "6";
       else if (num==5) return "5";
       else if (num==4) return "4";
       else if (num==3) return "3";
       else if (num==2) return "2";
       else if (num==1) return "1";
       else return "0";
}

function changeBackground2(number)
{
if(number == 1)
{
rnum1=rcount%16;
if (rcount <15){
rcount=rcount+1;
}
}
```

```
if(number == 2){
gnum1=gcount%16;
if (gcount <15){
gcount=gcount+1;
}
}

if(number == 3){
bnum1=bcount%16;
if (bcount <15){
bcount=bcount+1;
}
}

if(number == 4)
{rnum1=rcount%16;
if (rcount > 0){
rcount=rcount-1;
}
}

if(number == 5){
gnum1=gcount%16;
if (gcount > 0){
gcount=gcount-1;
}
}

if(number == 6){
bnum1=bcount%16;
if (bcount > 0){
bcount=bcount-1;
}
}

hexNumber2 = prefix+num2hex(rnum1)+num2hex(rnum2)+num2hex(gnum1)+num2hex(gnum2)+
    ➥num2hex(bnum1)+num2hex(bnum2);document.bgColor=hexNumber2

}

    function GoBack() {
        if (confirm("Are you sure you want to go to the previous page?")) {
                history.back()
        }
    }

    function GoForward() {
        if (confirm("Are you sure you want to go to the next page?")) {
                history.forward()
        }
    }

    function GoHome() {
        if (confirm("Are you sure you want to go to the Home page?")) {
                history.go('boots.com/~craigel/craig.htm')
        }
    }

  today = new Date()
    if(today.getMinutes() < 10){
        pad = "0"}
    else
    pad = "";
  document.write("<center><FONT SIZE=6>Welcome to My JavaScript Page</FONT></center>")
    if((today.getHours() < 12) && (today.getHours() >= 6)){
document.write("<center><FONT SIZE=4>Good Morning</FONT></center>")
}
    if((today.getHours() >= 12) && (today.getHours() < 18)){
document.write("<center><FONT SIZE=4>Good Afternoon</FONT></center>")
}
    if((today.getHours() >= 18) && (today.getHours() <= 23)){
document.write("<center><FONT SIZE=4>Good Evening</FONT></center>")
}
    if((today.getHours() >= 0) && (today.getHours() < 4)){
document.write("<center><FONT SIZE=4>You are up late, you should be in Bed</FONT></
➥center>")
}
    if((today.getHours() >= 4) && (today.getHours() <= 6)){
document.write("<center><FONT SIZE=4>Wow! You are up early, you should be in Bed</
➥FONT></center>")
}

  document.write("The time now is:",today.getHours(),":",pad,today.getMinutes())
  document.write("<br>The date is: ",
today.getMonth()+1,"/",today.getDate(),"/",today.getYear(),"<br>");

// end hiding contents -->

</script>

</head>

<body>
```

```
<hr>
<P>This document should only be viewed with Netscape 2.0 beta 5 and above.</P>
<IMG SRC="at_work.gif" BORDER=0>Page is under construction
<br>
<h3> Other JavaScript Pages <h3>
<h4>
<A HREF="time.htm"><IMG SRC="time.gif" BORDER=0>See JavaScript Clock</A><br>
<a HREF="banner.htm"><IMG SRC="banner.gif" BORDER=0>See JavaScript scrolling Banner
➥</a><br>
<a HREF="status.htm"><IMG SRC="status.gif" BORDER=0>See New JavaScript status line
➥Banner and clock</a>
</h4>

<br>
<p>
Please enter your name here First and then click outside the edit box:
</p>
<form>
  <input type="text" name="name" onBlur="getname(this.value)" value="">
</form>

<br>
Careful, Dont get to close to this icon<br>
<a href="" onMouseOver="hello()"><IMG SRC="icnques1.gif" ALT="???"></a>

<br>
<form>
<input type="button" name="Button1" value="Push me" onclick="WinOpen()">
</form>

<br>
<FORM>
<TABLE BORDER="3" CELLPADDING="3">
<tr><td Align=center colspan=2>History Buttons</td></tr>
<tr>
<td><INPUT TYPE="button" VALUE="Back Out" ONCLICK="GoBack()"></td>
<td><INPUT TYPE="button" VALUE="Go Forward" ONCLICK="GoForward()"></td>
</tr>
</table>
</form>

<p>Just a Stupid Link, look at status bar.
<a href="slink.htm" onMouseOver="window.status='Just another stupid link...'; return
➥true">link</a>
</P>

<br>
```

```
<br>
<br>
<FORM METHOD="POST" NAME="background">
<TABLE BORDER="3" CELLPADDING="3">
<TR>
<TD Align=center colspan=6>Background Color Changer</td></tr>
<tr>
<TD Align=center><INPUT TYPE="button" VALUE="red" ONCLICK="changeBackground
➥('#FF0000')"></TD>
<TD Align=center><INPUT TYPE="button" VALUE="green" ONCLICK="changeBackground
➥('#00FF00')"></TD>
<TD Align=center><INPUT TYPE="button" VALUE="blue" ONCLICK="changeBackground
➥('#0000FF')"></TD>
<TD Align=center><INPUT TYPE="button" VALUE="white" ONCLICK="changeBackground
➥('#FFFFFF')"></TD>
<TD Align=center><INPUT TYPE="button" VALUE="black" ONCLICK="changeBackground
➥('#000000')"></TD>
<TD Align=center><INPUT TYPE="button" VALUE="grey" ONCLICK="changeBackground
➥('#C0C0C0')"></TD>
</tr>
</TABLE>

<TABLE BORDER="3" CELLPADDING="3">
<TR>
<TD Align=center colspan=3>Variable Background Color Changer</td></tr>
<TR>
<TD Align=center><INPUT TYPE="button" VALUE="Increase Red" ONCLICK="
➥changeBackground2(1)"></TD>
<TD Align=center><INPUT TYPE="button" VALUE="Increase Green" ONCLICK="
➥changeBackground2(2)"></TD>
<TD Align=center><INPUT TYPE="button" VALUE="Increase Blue" ONCLICK="
➥changeBackground2(3)"></TD>
</tr>
<TR Align=center><TD><INPUT TYPE="button" VALUE="Decrease Red" ONCLICK="
➥changeBackground2(4)"></TD>
<TD Align=center><INPUT TYPE="button" VALUE="Decrease Green" ONCLICK="
➥changeBackground2(5)"></TD>
<TD Align=center><INPUT TYPE="button" VALUE="Decrease Blue" ONCLICK="
➥changeBackground2(6)"></TD>
</tr>
<TR>
<TD Align=center colspan=3>Keep pressing buttons increase color<br>
The color will start as black</td></tr>
<TR></TABLE>
</FORM>

<P>
```

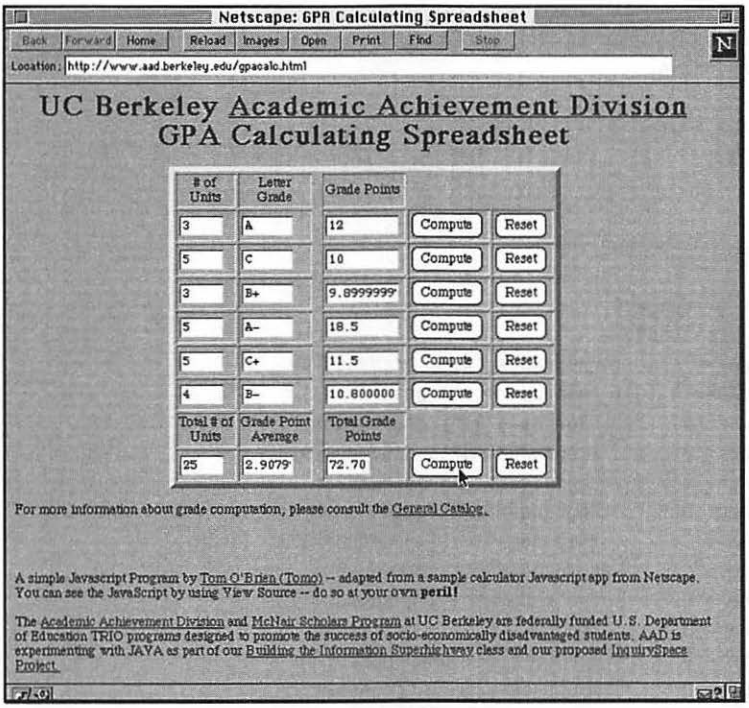
```
Much of the information that helped me produce this page came from the following sites:
<br>
<a href="http://ourworld.compuserve.com/homepages/vood/script.htm" TARGET="_top">
➥JavaScript by Voodoo</a>
<br>
<a
href="http://home.netscape.com/comprod/products/navigator/version_2.0/script/script_info/
➥index.html" TARGET="_top">Netscape JavaScript Authoring Guide</a>
<br>
<a href="http://www.c2.org/~andreww/javascript/" TARGET="_top">JavaScript Index</a>
</P>

<br>
Page Last Upadated:
  <script language="LiveScript">
  <!--  hide script from old browsers
    document.write(document.lastModified)
  // end hiding contents -->
  </script>

<br>
<hr>
<A HREF="mailto:Craigel@nando.net"><IMG SRC="mail.gif" ALT="Links"  HSPACE=1 BORDER=0></A>
<center>
<A HREF="craigf.htm"><IMG SRC="icnhome1.gif" ALT="Home Page" WIDTH=39 HEIGHT=39 HSPACE=1
➥BORDER=0 UNITS=pixels></a>
<A HREF="animate.htm"><IMG SRC="action.gif" ALT="Animations" WIDTH=39 HEIGHT=39 HSPACE=1
➥BORDER=0 UNITS=pixels></A>
<A HREF="images.htm"><IMG SRC="icnpall1.gif" ALT="Images" WIDTH=39 HEIGHT=39 HSPACE=1
➥BORDER=0 UNITS=pixels></A>
<A HREF="iresume.htm"><IMG SRC="resume.gif" ALT="Resume" WIDTH=39 HEIGHT=39 HSPACE=1
➥BORDER=0 UNITS=pixels></A>
<A HREF="links.htm"><IMG SRC="icnlink1.gif" ALT="Links" WIDTH=39 HEIGHT=39 HSPACE=1
➥BORDER=0 UNITS=pixels></A>
<A HREF="java2.htm"><IMG SRC="jnote.gif" ALT="Links" WIDTH=39 HEIGHT=39 HSPACE=1
➥BORDER=0 UNITS=pixels></A>
<A HREF="vrml.htm"><IMG SRC="vrml.gif" ALT="vrml" WIDTH=39 HEIGHT=39 HSPACE=1
➥BORDER=0 UNITS=pixels></A>
</center>

</body>
</html>
```

UC Berkeley GPA Calculator

http://www.aad.berkeley.edu/gpacalc.html

Tom O'Brien, tomo@uclink.berkeley.edu

Nice calculating spreadsheet for figuring grade point averages.


Netscape: GPA Calculating Spreadsheet
Back
Forward
Home
Reload
Images
Open
Print
Find
Stop
N
Location: http://www.aad.berkeley.edu/gpacalc.html

UC Berkeley Academic Achievement Division
GPA Calculating Spreadsheet

# of Units	Letter Grade	Grade Points		
3	A	12	Compute	Reset
5	C	10	Compute	Reset
3	B+	9.8999999	Compute	Reset
5	A-	18.5	Compute	Reset
5	C+	11.5	Compute	Reset
4	B-	10.800000	Compute	Reset
Total # of Units	Grade Point Average	Total Grade Points		
25	2.9079	72.70	Compute	Reset

For more information about grade computation, please consult the General Catalog.

A simple Javascript Program by Tom O'Brien (Tomo) -- adapted from a sample calculator Javascript app from Netscape. You can see the JavaScript by using View Source -- do so at your own **peril!**

The Academic Achievement Division and McNair Scholars Program at UC Berkeley are federally funded U.S. Department of Education TRIO programs designed to promote the success of socio-economically disadvantaged students. AAD is experimenting with JAVA as part of our Building the Information Superhighway class and our proposed InquirySpace Project.

```
<HTML>
<HEAD>
<TITLE>GPA Calculating Spreadsheet</TITLE>
<center><h1>UC Berkeley <a href=http://www.aad.berkeley.edu/uga/osl/aad/>Academic
➥Achievement Division</a> GPA Calculating Spreadsheet</h1></center>
<SCRIPT LANGUAGE="LiveScript">
<!-- hide this script tag's contents from old browsers
function checkNumber(input, min, max, msg)
{
    msg = msg + " field has invalid data: " + input.value;
    var str = input.value;
    for (var i = 0; i < str.length; i++) {
        var ch = str.substring(i, i + 1)
        if ((ch < "0" || "9" < ch) && ch != '.') {
            alert(msg);
            return false;
        }
    }
    var num = 0 + str
    if (num < min || max < num) {
        alert(msg + " not in range [" + min + ".." + max + "]");
        return false;
    }
    input.value = str;
    return true;
}

function computeGradenum2(input)
{
          var gradenum=0;
    var thegrade=input;
   if (thegrade=="A+" || thegrade=="a+") gradenum=4;
   if (thegrade=="A"  || thegrade=="a") gradenum=4;
  if (thegrade=="A-" || thegrade=="a-") gradenum=3.7;
  if (thegrade=="B+"  || thegrade=="b+") gradenum=3.3;
  if (thegrade=="B" || thegrade=="b") gradenum=3;
  if (thegrade=="B-" || thegrade=="b-") gradenum=2.7;
  if (thegrade=="C+"  || thegrade=="c+") gradenum=2.3;
  if (thegrade=="C"  || thegrade=="c") gradenum=2;
  if (thegrade=="C-"  || thegrade=="c-") gradenum=1.7;
  if (thegrade=="D+"  || thegrade=="d+") gradenum=1.3;
  if (thegrade=="D" || thegrade=="d") gradenum=1;
  if (thegrade=="D-"  || thegrade=="d-") gradenum=.7;
  if (thegrade=="F+" || thegrade=="f+" ) gradenum=.3;
  if (thegrade=="F" || thegrade=="f" ) gradenum=0;
  if (thegrade=="F-" || thegrade=="f-") gradenum=0;
     return gradenum;
     }

function computeField(input)
{
    if (input.value != null && input.value.length != 0)
        input.value = "" + eval(input.value);
        computeForm(input.form);
}
```

```
function computeForm(form)
{
    if ((form.units.value == null || form.units.value.length == 0)) {
        return;
    }
    if (!checkNumber(form.units, .5, 10, "Units")) {
        form.gradepoints.value = "Invalid";
        return;
    }
    if ((form.grade.value == null || form.grade.value.length == 0)) {
        return;
    }
    form.gradepoints.value = ((computeGradenum2(form.grade.value)) *
form.units.value);
}

function computesumForm(form)
{
      document.forms[6].gradepoints.value=0;
      document.forms[6].units.value=0;
      document.forms[6].grade.value=0;

     for(var i=0; i<6; i++) {
        if (!(document.forms[i].units.value == null || document.forms[i].units.
        ➥value.length == 0)) {
           if (!(document.forms[i].units.value == null || document.forms[i].units.
           ➥value.length == 0)) {
                var temp=computeField(document.forms[i].gradepoints);
                var temp=computeField(document.forms[i].units);
                var temp=computeForm(document.forms[i]);
           if (!(document.forms[i].gradepoints.value == 0))
           document.forms[6].gradepoints.value =
eval(document.forms[6].gradepoints.value)+(eval(document.forms[i].gradepoints.value));

           if (!(document.forms[i].units.value == 0))
           document.forms[6].units.value =
eval(document.forms[6].units.value)+(eval(document.forms[i].units.value));
           }
      }
   }
if (!(document.forms[6].units.value == 0))
document.forms[6].grade.value=(eval(document.forms[6].gradepoints.value)/
➥(eval(document.forms[6].units.value)));

}
```

```
function clearForm(form)
{
    form.units.value = "";
    form.grade.value = "";
    form.gradepoints.value = "";
}
<!-- done hiding from old browsers -->
</SCRIPT>
</HEAD>
<FONT SIZE=3>
<CENTER>

<TABLE border=4>
<FORM method=POST>
<TR>
<TD><DIV ALIGN=CENTER>  # of<br>Units</DIV></TD>
<TD><DIV ALIGN=CENTER>Letter<br>Grade</DIV></TD>
<TD> </TD>
<TD><DIV ALIGN=CENTER> Grade Points</DIV></TD>
</TR>
<TR>
<TD><INPUT TYPE=TEXT NAME=units  SIZE=5 onChange=computeField(this)> </TD>
<TD><INPUT TYPE=TEXT NAME=grade  SIZE=6 onChange=computeForm(this.form)> </TD>
<TD> </TD>
<TD><INPUT TYPE=TEXT NAME=gradepoints   SIZE=9 onChange=computeField(this)>
</TD>
<TD><INPUT TYPE="button" VALUE="Compute"   onClick=computeForm(this.form)> </TD>
<TD><INPUT TYPE="reset"  VALUE="Reset"     onClick=clearForm(this.form)> </TD>
</TR>
</FORM>
<FORM method=POST>
<TR>
<TD><INPUT TYPE=TEXT NAME=units  SIZE=5 onChange=computeField(this)> </TD>
<TD><INPUT TYPE=TEXT NAME=grade  SIZE=6 onChange=computeForm(this.form)> </TD>
<TD> </TD>
<TD><INPUT TYPE=TEXT NAME=gradepoints   SIZE=9 onChange=computeField(this)>
</TD>
<TD><INPUT TYPE="button" VALUE="Compute"   onClick=computeForm(this.form)> </TD>
<TD><INPUT TYPE="reset"  VALUE="Reset"     onClick=clearForm(this.form)> </TD>
</TR>
</FORM>
<FORM method=POST>
<TR>
<TD><INPUT TYPE=TEXT NAME=units  SIZE=5 onChange=computeField(this)> </TD>
<TD><INPUT TYPE=TEXT NAME=grade  SIZE=6 onChange=computeForm(this.form)> </TD>
<TD> </TD>
<TD><INPUT TYPE=TEXT NAME=gradepoints   SIZE=9 onChange=computeField(this)>
</TD>
<TD><INPUT TYPE="button" VALUE="Compute"   onClick=computeForm(this.form)> </TD>
<TD><INPUT TYPE="reset"  VALUE="Reset"     onClick=clearForm(this.form)> </TD>
</TR></FORM>
```

```
<FORM method=POST>
<TR>
<TD><INPUT TYPE=TEXT NAME=units  SIZE=5 onChange=computeField(this)> </TD>
<TD><INPUT TYPE=TEXT NAME=grade  SIZE=6 onChange=computeForm(this.form)> </TD>
<TD> </TD>
<TD><INPUT TYPE=TEXT NAME=gradepoints   SIZE=9 onChange=computeField(this)>
</TD>
<TD><INPUT TYPE="button" VALUE="Compute"   onClick=computeForm(this.form)> </TD>
<TD><INPUT TYPE="reset"  VALUE="Reset"     onClick=clearForm(this.form)> </TD>
</TR>
</FORM>
<FORM method=POST>
<TR>
<TD><INPUT TYPE=TEXT NAME=units  SIZE=5 onChange=computeField(this)> </TD>
<TD><INPUT TYPE=TEXT NAME=grade  SIZE=6 onChange=computeForm(this.form)> </TD>
<TD> </TD>
<TD><INPUT TYPE=TEXT NAME=gradepoints   SIZE=9 onChange=computeField(this)>
</TD>
<TD><INPUT TYPE="button" VALUE="Compute"   onClick=computeForm(this.form)> </TD>
<TD><INPUT TYPE="reset"  VALUE="Reset"     onClick=clearForm(this.form)> </TD>
</TR>
</FORM>
<FORM method=POST>
<TR>
<TD><INPUT TYPE=TEXT NAME=units  SIZE=5 onChange=computeField(this)> </TD>
<TD><INPUT TYPE=TEXT NAME=grade  SIZE=6 onChange=computeForm(this.form)> </TD>
<TD> </TD>
<TD><INPUT TYPE=TEXT NAME=gradepoints   SIZE=9 onChange=computeField(this)>
</TD>
<TD><INPUT TYPE="button" VALUE="Compute"   onClick=computeForm(this.form)> </TD>
<TD><INPUT TYPE="reset"  VALUE="Reset"     onClick=clearForm(this.form)> </TD>
</TR>
</FORM>
<TR>
<TD><DIV ALIGN=CENTER>  Total # of<br>Units</DIV></TD>
<TD><DIV ALIGN=CENTER>Grade Point<br>Average</DIV></TD>
<TD> </TD>
<TD><DIV ALIGN=CENTER>Total Grade <br>Points</DIV></TD>
</TR>
<FORM method=POST>
<TR>
<TD><INPUT TYPE=TEXT NAME=units  SIZE=5> </TD>
<TD><INPUT TYPE=TEXT NAME=grade  SIZE=6> </TD>
<TD> </TD>
<TD><INPUT TYPE=TEXT NAME=gradepoints   SIZE=5></TD>
<TD><INPUT TYPE="button" VALUE="Compute"   onClick=computesumForm(this.form)> </TD>
<TD><INPUT TYPE="reset"  VALUE="Reset"     onClick=clearForm(this.form)> </TD>
</TR>
</form>
</table>
```

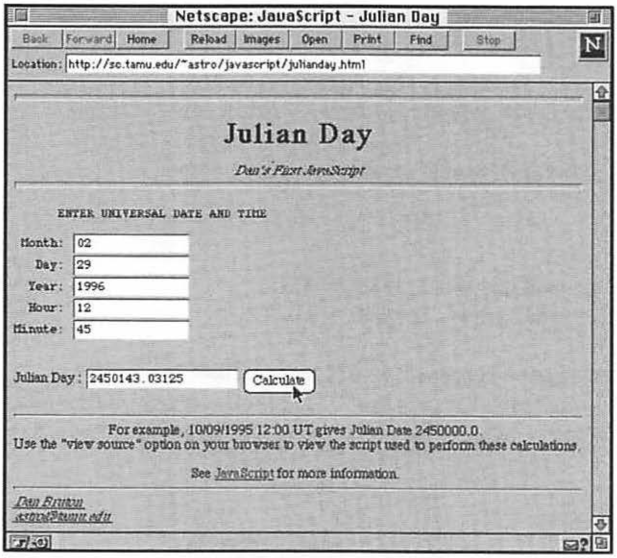
```
</CENTER>
<FONT SIZE=3><p>
For more information about grade computation, please consult the <a href=http://
➥www.urel.berkeley.edu/UREL_1/Catalog95/CatChapSix/6-3.html>General Catalog.</a>
<pre>

</pre>
A simple JavaScript Program by <a href=mailto:tomo@uclink.berkeley.edu>Tom O'Brien
(Tomo)</a> -- adapted from a sample calculator JavaScript app from Netscape.  You can
see the JavaScript by using View Source -- do so at your own <b>peril!</b><p>The <a
href=http://www.aad.berkeley.edu/uga/osl/aad/>Academic Achievement Division</a> and <a
href=http://www.aad.berkeley.edu/uga/osl/mcnair/>McNair Scholars Program</a> at UC
Berkeley are federally funded U.S. Department of Education TRIO programs designed to
promote the success of socio-economically disadvantaged students.  AAD is experiment-
ing with JAVA as part of our <a href=http://www.aad.berkeley.edu/UGA/OSL/AAD/AADClass/
computer2.html>Building the Information Superhighway</a> class and our proposed <a
href=http://www.aad.berkeley.edu/inquiryspace/>InquirySpace Project.</a>
</BODY>
</HTML>
```

Julian Day

http://sc.tamu.edu/~astro/javascript/julianday.html

Dan Bruton, astro@tamu.edu

Astronomy calculations.



Location: http://sc.tamu.edu/~astro/javascript/julianday.html

Julian Day

Dan's First JavaScript

ENTER UNIVERSAL DATE AND TIME

Month: 02
Day: 29
Year: 1996
Hour: 12
Minute: 45

Julian Day: 2450143.03125 Calculate

For example, 10/09/1995 12:00 UT gives Julian Date 2450000.0.
Use the "view source" option on your browser to view the script used to perform these calculations.

See JavaScript for more information.

Dan Bruton
astro@tamu.edu

```
<HTML>
<HEAD>
<title>JavaScript - Julian Day</title>

<SCRIPT LANGUAGE="JavaScript">
<!-- hide this script tag's contents from old browsers
function compute(form) {
    MM=eval(form.nmonth.value)
    DD=eval(form.nday.value)
    YY=eval(form.nyear.value)
    HR=eval(form.nhour.value)
    MN=eval(form.nminute.value)
    with (Math) {
      HR = HR + (MN / 60);
      GGG = 1;
      if (YY <= 1585) GGG = 0;
      JD = -1 * floor(7 * (floor((MM + 9) / 12) + YY) / 4);
      S = 1;
      if ((MM - 9)<0) S=-1;
      A = abs(MM - 9);
      J1 = floor(YY + S * floor(A / 7));
      J1 = -1 * floor((floor(J1 / 100) + 1) * 3 / 4);
      JD = JD + floor(275 * MM / 9) + DD + (GGG * J1);
      JD = JD + 1721027 + 2 * GGG + 367 * YY - 0.5;
      JD = JD + (HR / 24);
    }
    form.result.value = JD;
}
// done hiding from old browsers -->
</SCRIPT>
</HEAD>

<BODY>
<center>
<hr size=5>
<h1>Julian Day</h1>
<i>Dan's First JavaScript</i>
<hr size=5>
</center>
<FORM>
<pre>
      ENTER UNIVERSAL DATE AND TIME
```

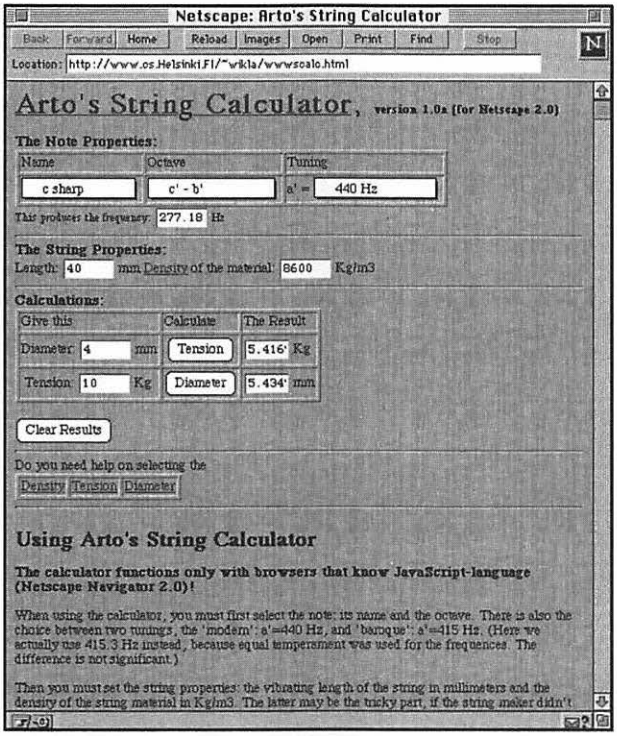
```
 Month: <INPUT TYPE="text" NAME="nmonth" SIZE=15>
   Day: <INPUT TYPE="text" NAME="nday" SIZE=15>
  Year: <INPUT TYPE="text" NAME="nyear" SIZE=15>
  Hour: <INPUT TYPE="text" NAME="nhour" SIZE=15>
Minute: <INPUT TYPE="text" NAME="nminute" SIZE=15>
</pre>

<BR>
Julian Day :
<INPUT TYPE="text" NAME="result" SIZE=20>
<INPUT TYPE="button" VALUE="Calculate" ONCLICK="compute(this.form)">
<BR>
</FORM>
<hr>
<center>
For example, 10/09/1995 12:00 UT gives Julian Date 2450000.0.
<br>Use the "view source" option on your browser to view the script used to
perform these calculations.  <br>See <a href="http://home.netscape.com/comprod/
➥products/navigator/version_2.0/script/script_info/index.html">JavaScript</a> for more
➥information.
</center>
<hr>
<address>
<a href="http://www.isc.tamu.edu/~astro/">Dan Bruton</a>
<br><a href="mailto:astro@tamu.edu">astro@tamu.edu</a>
</address>
</BODY>
</HTML>
```

Arto's String Calculator

http://www.cs.Helsinki.FI/~wikla/wwwscalc.html

Arto Wikla, Arto.Wikla@cs.Helsinki.FI

A calculator for (musical instrument) string calculations: diameter -> tension and tension -> diameter.


Netscape: Arto's String Calculator
Back
Forward
Home
Reload
Images
Open
Print
Find
Stop
N
Location: http://www.cs.Helsinki.FI/~wikla/wwwscalc.html

Arto's String Calculator, **version 1.0a (for Netscape 2.0)**

The Note Properties:

Name	Octave	Tuning
c sharp	c' - b'	a' = 440 Hz

This produces the frequency: 277.18 Hz

The String Properties:

Length: 40 mm Density of the material: 8600 Kg/m3

Calculations:

Give this	Calculate	The Result
Diameter: 4 mm	Tension	5.416' Kg
Tension: 10 Kg	Diameter	5.434' mm

Clear Results

Do you need help on selecting the

Density Tension Diameter

Using Arto's String Calculator

The calculator functions only with browsers that know JavaScript-language (Netscape Navigator 2.0)!

When using the calculator, you must first select the note: its name and the octave. There is also the choice between two tunings, the 'modern': a'=440 Hz, and 'baroque': a'=415 Hz. (Here we actually use 415.3 Hz instead, because equal temperament was used for the frequences. The difference is not significant.)

Then you must set the string properties: the vibrating length of the string in millimeters and the density of the string material in Kg/m3. The latter may be the tricky part, if the string maker didn't

```
<HEAD>
<TITLE>Arto's String Calculator, v1.0a</TITLE>

<!-- Copyright 1996 by Arto Wikla, University of Helsinki, Finland-->
<!-- Free permission is given to re-use this code, if and only if -->
<!-- this copyright notice is kept in. -->

<SCRIPT LANGUAGE="JavaScript">
<!--   COPYRIGHT Arto Wikla, 1996, Arto.Wikla@cs.helsinki.fi

function MakeArray(n) {
   this.length = n;
   for (var i = 1; i <= n; i++) {
     this[i] = 0 }
     return this
 }
```

```
var F = new MakeArray(13)
  F[0]  = 246.94
  F[1]  = 261.63
  F[2]  = 277.18
  F[3]  = 293.66
  F[4]  = 311.13
  F[5]  = 329.63
  F[6]  = 349.23
  F[7]  = 369.99
  F[8]  = 392
  F[9]  = 415.3
  F[10] = 440
  F[11] = 466.16
  F[12] = 493.88

var OktAla = new MakeArray(9)
    OktAla[0] = 0.0625
    OktAla[1] = 0.125
    OktAla[2] = 0.25
    OktAla[3] = 0.5
    OktAla[4] = 1
    OktAla[5] = 2
    OktAla[6] = 4
    OktAla[7] = 8
    OktAla[8] = 16

function Herziluku(form)
 {return F[form.Note.selectedIndex+form.a415.selectedIndex]
        * OktAla[form.Octave.selectedIndex]
 }

function Veto(F,L,D,P)
 {return (F*L*D)*(F*L*D)*(3.14159265*P/(9.81*1000000*1000000)) }

function Paksuus(T,F,L,P)
  {return Math.sqrt((1000000/F)*(1000000/F)* T*9.81/L/L/3.14159265/P) }

function LaskeVeto(form)
  {var F = form.herzit.value   // Herziluku(form)
   var L = form.StLen.value
   var P = form.StDen.value
   var D = form.StThiIn.value
   form.StTenOut.value = Math.round(1000*Veto(F,L,D,P))/1000
  }
```

```
function LaskePaksuus(form)
  {var F = form.herzit.value     // Herziluku(form)
   var L = form.StLen.value
   var P = form.StDen.value
   var T = form.StTenIn.value
   form.StThiOut.value = Math.round(1000*Paksuus(T,F,L,P))/1000
  }

function SiivoaTul(form)
  {form.StTenOut.value = ""
   form.StThiOut.value = ""  }

function Tarkista(input, min, max, msg)
{   msg = msg + " field has invalid data: " + input.value;
    var str = input.value;
    if (str.length == 0)
     { alert(msg);
       return false;  }
    for (var i = 0; i < str.length; i++)
     {  var ch = str.substring(i, i + 1)
        if ((ch < "0" ¦¦ "9" < ch) && ch != '.') {
            alert(msg);
            return false; }
     }
    var num = parseFloat(str)
    if (num < min ¦¦ max < num)
     { alert(msg + " not in range [" + min + ".." + max + "]");
       return false; }
    input.value = str;
    return true;
}

function TarkistaLomake(form)
{ return Tarkista(form.StLen,10 , 4000, 'Length')     &&
         Tarkista(form.StDen,500 , 10000, 'Density')
}

<!--  -->
</SCRIPT>

</HEAD>
<BODY BACKGROUND="puu.gif"  LINK="#800000" VLINK="#000080" ALINK="#3FFF3F">

<FORM>
```

```
<A NAME="TOP">
<H1><A HREF=#P0>Arto's String Calculator</A>,
    <FONT SIZE=-1>version 1.0a (for Netscape 2.0)</FONT></H1></A>

<STRONG>The Note Properties:</STRONG>

<TABLE border>

<TR>
<TD>Name</TD>
<TD>Octave</TD>
<TD>Tuning</TD>
</TR>

<TR>

<TD>
<SELECT NAME="Note"
   onChange="SiivoaTul(this.form); herzit.value=Herziluku(this.form)">
<OPTION> c
<OPTION> c sharp
<OPTION> d
<OPTION> e flat
<OPTION> e
<OPTION> f
<OPTION> f sharp
<OPTION> g
<OPTION> g sharp
<OPTION SELECTED> a
<OPTION> b flat
<OPTION> b
</SELECT>
</TD>

<TD>
<SELECT NAME="Octave"
   onChange="SiivoaTul(this.form); herzit.value=Herziluku(this.form)">
<OPTION> C'' - B''
<OPTION> C' - B'
<OPTION> C - B
<OPTION> c - b
<OPTION SELECTED> c' - b'
<OPTION> c'' - b''
<OPTION> c''' - b'''
<OPTION> c'''' - b''''
<OPTION> c''''' - b'''''
</SELECT>
</TD>
```

```
<TD>
a' =
<SELECT NAME="a415"
   onChange="SiivoaTul(this.form); herzit.value=Herziluku(this.form)">
<OPTION> 415,3 Hz
<OPTION SELECTED> 440 Hz
</SELECT>
</TD>

</TR>

</TABLE>

<FONT SIZE=-1>
This produces the frequency:

<INPUT SIZE=6  NAME="herzit" VALUE="440"
  onChange="SiivoaTul(this.form);
            alert('Do not edit the frequency!');
            herzit.value=Herziluku(this.form)"> Hz
<FONT>
<HR>

<STRONG>The String Properties:</STRONG><BR>

Length:
<INPUT SIZE=6 NAME="StLen" VALUE=""
 onChange="SiivoaTul(this.form);
           if (! Tarkista(StLen,10 , 4000, 'Length'))
              StLen.select()"> mm

<A HREF=#P1>Density</A> of the material:
<INPUT SIZE=6 NAME="StDen" VALUE=""
onChange="SiivoaTul(this.form);
          if (!Tarkista(StDen,500 , 10000, 'Density'))
             StDen.select()"> Kg/m"

<HR>
<STRONG>Calculations:</STRONG>

<TABLE border>
<TR>
<TD>Give this</TD>
<TD>Calculate</TD>
<TD>The Result</TD>
</TR>
<TR>
<TD><CENTER>
Diameter:
```

```
<INPUT SIZE=6 NAME="StThiIn" VALUE=""
 onChange="StTenOut.value='';
          if (! Tarkista(StThiIn,0.01, 5.0, 'Diameter'))
              StThiIn.select()"> mm</CENTER></TD>

<TD><CENTER>
<INPUT TYPE="button" NAME="result1" VALUE="Tension"
  OnClick=" if ( TarkistaLomake(this.form) &&
                 Tarkista(form.StThiIn,0.01, 5.0, 'Diameter') )
                 LaskeVeto(this.form)"></CENTER></TD>

<TD>
<INPUT SIZE=5 NAME="StTenOut" VALUE=""
  onChange='StTenOut.value='';
           alert('Do not edit the result!')"> Kg</TD>
</TR>

<TR>
<TD><CENTER>
Tension:
<INPUT SIZE=6 NAME="StTenIn" VALUE=""
 onChange="StThiOut.value='';
          if (! Tarkista(StTenIn,0.01, 100, 'Tension'))
             StTenIn.select()"> Kg</CENTER></TD>

<TD><CENTER>
<INPUT TYPE="button" NAME="result2" VALUE="Diameter"
  OnClick=" if ( TarkistaLomake(this.form) &&
                 Tarkista(StTenIn,0.01, 100, 'Tension') )
                 LaskePaksuus(this.form)"></CENTER></TD>

<TD><INPUT SIZE=5 NAME="StThiOut" VALUE=""
  onChange='StThiOut.value='';
           alert('Do not edit the result!')"> mm</TD>
</TR>
</TABLE>

<P>
 <INPUT TYPE="button"
      VALUE="Clear Results"
      OnClick="SiivoaTul(this.form)">

<hr>
```

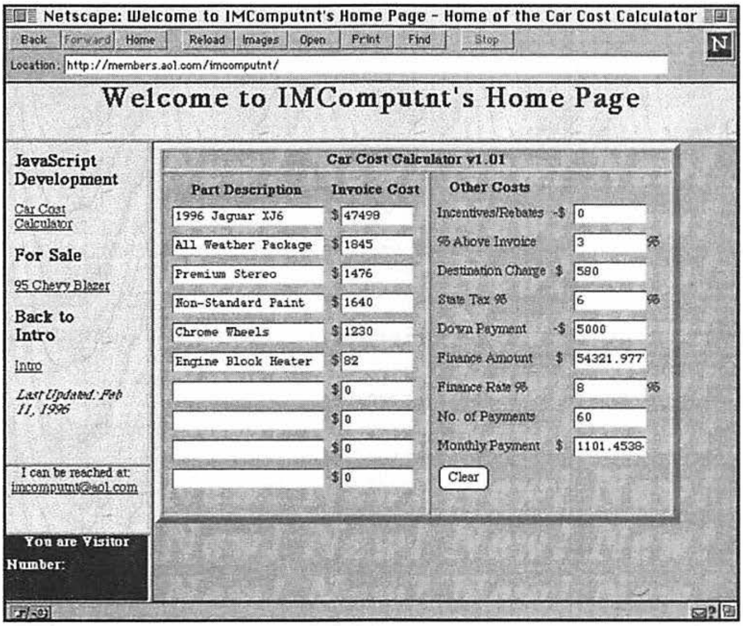
We have not included the rest of the straight HTML here, because the JavaScript portion is what we care about.

Car Cost Calculator

http://members.aol.com/imcomputnt/

Matthew J Graci, imcomputnt@aol.com

A calculator for calculating monthly payments for cars you can't afford.


Netscape: Welcome to IMComputnt's Home Page - Home of the Car Cost Calculator
Back
Forward
Home
Reload
Images
Open
Print
Find
Stop
N
Location: http://members.aol.com/imcomputnt/

Welcome to IMComputnt's Home Page

JavaScript Development

Car Cost Calculator

For Sale

95 Chevy Blazer

Back to Intro

Intro

Last Updated: Feb 11, 1996

I can be reached at: imcomputnt@aol.com

You are Visitor Number:

Car Cost Calculator v1.01

Part Description		Invoice Cost
1996 Jaguar XJ6	$	47498
All Weather Package	$	1845
Premium Stereo	$	1476
Non-Standard Paint	$	1640
Chrome Wheels	$	1230
Engine Block Heater	$	82
	$	0
	$	0
	$	0
	$	0

Other Costs			
Incentives/Rebates	-$	0	
% Above Invoice		3	%
Destination Charge	$	580	
State Tax %		6	%
Down Payment	-$	5000	
Finance Amount	$	54321.977	
Finance Rate %		8	%
No. of Payments		60	
Monthly Payment	$	1101.4538	

```
<HTML>
<HEAD><TITLE>Car Cost Calculator</TITLE>

<SCRIPT>
<!-- hide this script tag's contents from old browsers

//Car Cost Calculator
//Written by Matthew J Graci

function checkNumber(input,form1,form2)
{
 if(input.value=='')
        {input.value=input.defaultValue
         input.focus()
         input.select()
        }
        status=''
    msg ="This field requires numeric data: " + input.value;

    var str = input.value;
    for (var i = 0; i < str.length; i++)
                {var ch = str.substring(i, i + 1)
        if ((ch < "0" ¦¦ "9" < ch) && ch != '.') {
                        input.value=input.defaultValue
                        input.focus()
                        input.select()
                        status=msg}
    }
    add_input(form1,form2)
 }
function add_input(form1,form2)
{    var total1,total2,total3,total4,total5,total6


total1=(form1.cost1.value*1+form1.cost2.value*1+form1.cost3.value*1+form1.cost4.
➥value*1+form1.cost5.value*1)

total2=(total1*1+form1.cost6.value*1+form1.cost7.value*1+form1.cost8.value*1+form1.
➥cost9.value*1+form1.cost10.value*1)
         total3=(total2-form2.incentives.value*1)
         total4=(total3*1 + (('.01' * form2.above_invoice.value)* total3))
         total5=(total4*1+form2.destination.value*1)
         total6=(total5*1 + (('.01' * form2.state_tax.value)* total5))
         form2.total_cost.value=(total6*1 -form2.down_payment.value)
         compute_payments(form2)
}
function compute_payments(form2)
{
    var i = form2.interest_rate.value;
    if (i > 1.0) {
        i = i / 100.0;
        }
    i /= 12;

    var pow = 1;
    for (var j = 0; j < form2.no_of_payments.value; j++)
        pow = pow * (1 + i);
    form2.monthly_payments.value = (form2.total_cost.value * pow * i) / (pow - 1)
}
function selectField(field)
{
        field.select()
}
function clearForm(form1,form2)
{
        form1.desc1.value = ""
        form1.cost1.value = "0"
        form1.desc2.value = ""
        form1.cost2.value = "0"
        form1.desc3.value = ""
        form1.cost3.value = "0"
        form1.desc4.value = ""
        form1.cost4.value = "0"
        form1.desc5.value = ""
        form1.cost5.value = "0"
        form1.desc6.value = ""
        form1.cost6.value = "0"
        form1.desc7.value = ""
        form1.cost7.value = "0"
        form1.desc8.value = ""
        form1.cost8.value = "0"
        form1.desc9.value = ""
        form1.cost9.value = "0"
        form1.desc10.value = ""
        form1.cost10.value = "0"
        form2.incentives.value = "0"
        form2.above_invoice.value="3"
        form2.destination.value="0"
        form2.down_payment.value = "0"
        form2.total_cost.value="0"
        form2.interest_rate.value="8"
        form2.no_of_payments.value="60"
        form2.monthly_payments.value="0"
 }
<!-- done hiding from old browsers -->
</SCRIPT>
</HEAD>

<BODY BACKGROUND="newback.jpg" >
<TABLE Border=5>
<TR><TH Colspan=2>Car Cost Calculator v1.01</TH></TR>
<TR><TD>
<FORM NAME="form1">
        <TABLE>
        <TR>
        <TH>Part Description</TH><TH>Invoice Cost</TH>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc1 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost1 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc2 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost2 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc3 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost3 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc4 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost4 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc5 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost5 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc6 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost6 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc7 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost7 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc8 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost8 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc9 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost9 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=TEXT NAME=desc10 SIZE=20 onFocus="selectField(this)"></TD>
        <TD>$<INPUT TYPE=TEXT NAME=cost10 VALUE=0 SIZE=9 onFocus="selectField(this)"
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        </TABLE>
```

```
</FORM>
</TD>
<TD>
<FORM NAME="form2">
        <TABLE>
        <TR>
        <TH>Other Costs</TH>
        </TR>
        <TR>
        <TD> Incentives/Rebates</TD>
        <TD align=right>-$</TD>
        <TD><INPUT TYPE=TEXT NAME=incentives VALUE=0 SIZE=9
➥onFocus="selectField(this)" onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD> % Above Invoice</TD>
        <TD></TD>
        <TD><INPUT TYPE=TEXT NAME=above_invoice VALUE=3 SIZE=9
➥onFocus="selectField(this)" onChange="checkNumber(this,form1,form2)">% </TD>
        </TR>
        <TR>
        <TD> Destination Charge</TD>
        <TD align=right>$</TD>
        <TD><INPUT TYPE=TEXT NAME=destination VALUE=0 SIZE=9
➥onFocus="selectField(this)" onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD> State Tax %</TD>
        <TD></TD>
        <TD><INPUT TYPE=TEXT NAME=state_tax VALUE=6 SIZE=9 onFocus="selectField(this)"
➥onChange="checkNumber(this,form1,form2)">% </TD>
        </TR>
        <TR>
        <TD> Down Payment</TD>
        <TD align=right>-$</TD>
        <TD><INPUT TYPE=TEXT NAME=down_payment VALUE=0 SIZE=9
➥onFocus="selectField(this)" onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD> Finance Amount</TD>
        <TD align=right>$</TD>
        <TD><INPUT TYPE=TEXT NAME=total_cost VALUE=0 SIZE=9
➥onChange="checkNumber(this,form1,form2)"> </TD>
```

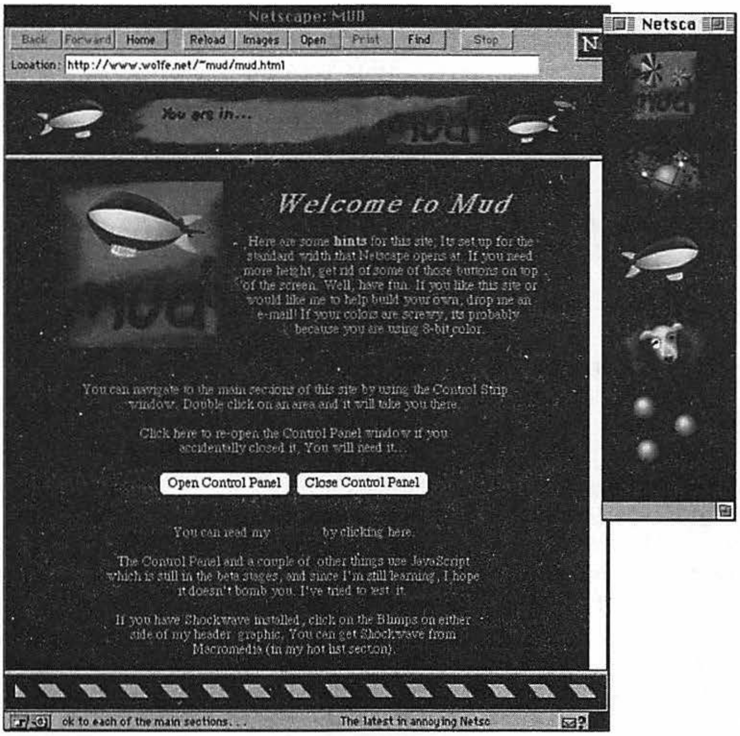
```
        </TR>
        <TR>
        <TD> Finance Rate % </TD>
        <TD></TD>
        <TD><INPUT TYPE=TEXT NAME=interest_rate VALUE=8 SIZE=9
➥onFocus="selectField(this)" onChange="checkNumber(this,form1,form2)">% </TD>
        </TR>
        <TR>
        <TD> No. of Payments</TD>
        <TD></TD>
        <TD><INPUT TYPE=TEXT NAME=no_of_payments VALUE=60 SIZE=9
➥onFocus="selectField(this)" onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD> Monthly Payment</TD>
        <TD align=right>$</TD>
        <TD><INPUT TYPE=TEXT NAME=monthly_payments VALUE=0 SIZE=9
onChange="checkNumber(this,form1,form2)"> </TD>
        </TR>
        <TR>
        <TD><INPUT TYPE=Button Value=Clear Name=Clear
➥onClick="clearForm(form1,form2)"> </TD>
        <TD></TD>
        </TR>
</TD>
</TABLE>
</FORM>
</TABLE>
</BODY>
</HTML>
```

MUD

http://www.wolfenet.com/~mud

Gordon Mueller, mud@gonzo.wolfenet.com

Floating control panel in its own window, scrolling text in status bar.



Welcome to Mud

Here are some **hints** for this site; Its set up for the standard width that Netscape opens at. If you need more height, get rid of some of those buttons on top of the screen. Well, have fun. If you like this site or would like me to help build your own, drop me an e-mail! If your colors are screwy, its probably because you are using 8-bit color.

You can navigate to the main sections of this site by using the Control Strip window. Double click on an area and it will take you there.

Click here to re-open the Control Panel window if you accidentally closed it, You will need it...

You can read my by clicking here.

The Control Panel and a couple of other things use JavaScript which is still in the beta stages, and since I'm still learning, I hope it doesn't bomb you. I've tried to test it.

If you have Shockwave installed, click on the Blimps on either side of my header graphic. You can get Shockwave from Macromedia (in my hot list secton).

```
<HTML><HEAD>
<TITLE>index.html</TITLE>
<SCRIPT language="LiveScript">
function windowOpener(){

controlWindow=window.open("../controlmap/controlmap.html","Control","toolbar=no,
➥location=no,directories=no,status=no,menubar=no,scrollbars=no,resizable=no,
➥width=110,height=385");

controlWindow=window.open("../controlmap/controlmap.html","Control","toolbar=no,
➥location=no,directories=no,status=no,menubar=no,scrollbars=no,resizable=no,
➥width=110,height=385");

}
</SCRIPT>
</HEAD>

<BODY BGCOLOR="#000000" TEXT="#BDBDBD" LINK="#1916A6" ALINK="#353136" VLINK="#353136"
onLoad="windowOpener()">
<MAP NAME="control.map">
<AREA SHAPE=RECT HREF="../mainpage/mainpage.html" COORDS="11,3,70,64" TARGET="main">
<AREA SHAPE=RECT HREF="../starpage/starpage.html" COORDS="4,82,79,142" TARGET="main">
```

```
<AREA SHAPE=RECT HREF="../blimp/blimp.html" COORDS="2,153,73,201" TARGET="main">
<AREA SHAPE=RECT HREF="../swamp/swamp.html" COORDS="6,215,78,274" TARGET="main">
<AREA SHAPE=RECT HREF="../forest/forest.html" COORDS="12,286,72,346" TARGET="main">
</MAP>
<P>
<P>

</P><CENTER><TABLE CELLPADDING="4"><TR><TD><CENTER><FONT SIZE=4>
<IMG SRC="../blimpbig.gif" ALIGN=LEFT>
</FONT></CENTER>
<H1><P ALIGN=Center>

</H1></TD><TD><H1 ALIGN=Center><I>Welcome to Mud</I>

</H1><P ALIGN=Center>
Here are some <B>hints</B> for this site; Its set up for the<BR>
standard width that Netscape opens at. If you need<BR>
more height, get rid of some of those buttons on top<BR>
of the screen. Well, have fun. If you like this site or <BR>
would like me to help build your own, drop me an <BR>e-mail! If your colors
are screwy, its probably <BR>because you are using 8-bit color.<BR>

</TD></TR></TABLE></CENTER><P>
</P><CENTER><TABLE CELLPADDING="4"><TR VALIGN="Top">
<TD COLSPAN=4>
<CENTER>You can
navigate to the main sections of this site by using the Control Strip <BR>
window. Double click on an area and it will take you there.<BR>

<BR>Click here to re-open the Control Panel window if you <BR>
accidentally closed it, You will need it...

<FORM>
<INPUT type="button" value="Open Control Panel" onClick="windowOpener()"
>
<INPUT type="button" value="Close Control Panel" onClick="controlWindow.close()"
>

</FORM><P><BR>
You can read my <A HREF="../reshtml/resume.html">Resume</A> by clicking here.<BR><BR>
The Control Panel and a couple of&nbsp;other things use JavaScript <BR>
which is still in the beta stages, and since I'm still learning, I hope <BR>
it doesn't bomb you. I've tried to test&nbsp;it.
<P>If you have Shockwave installed, click on the Blimps on either <BR>
side of my header&nbsp;graphic. You can get Shockwave from <BR>
Macromedia (in my hot list section).<BR>
</CENTER>
</TD></TR><TR><TD>
</TD><TD><BR>
```

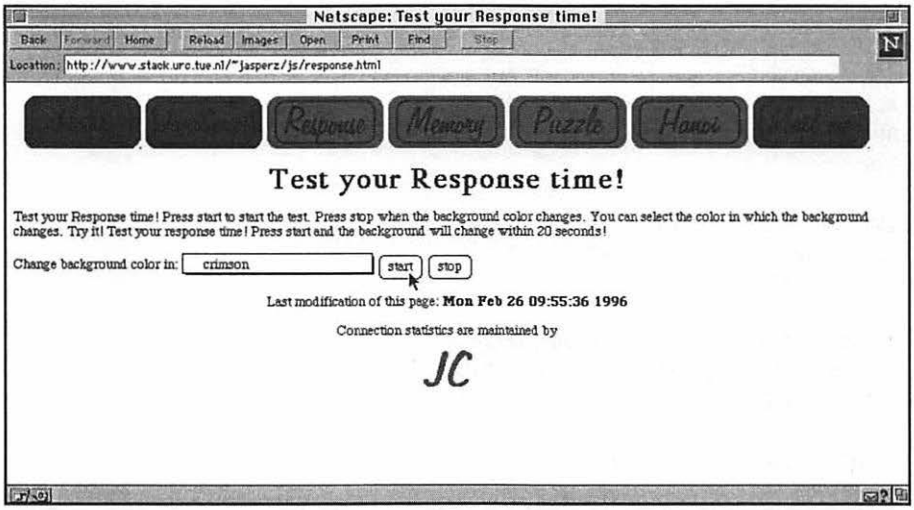
```
</TD><TD>
</TD></TR><TR><TD ROWSPAN=5>
</TD><TD ROWSPAN=5 VALIGN="Middle"><P ALIGN=Center>
<IMG ALIGN=Middle SRC="../controlmap/maptall.gif" USEMAP="#control.map" BORDER=0
➥WIDTH="85" HEIGHT="348">
</TD><TD><P>

<P ALIGN=Left><H2><I>Main</I></H2>
<p>This is where you are now.<BR>
</TD></TR><TR><TD><P>

<P><BR><H2><I>Inner and Outer Space</I></H2>
<p>Look, up in the sky its. . . an alien?<BR>
</TD></TR><TR><TD><P>

<P><BR><H2><I>Zeppelin</I></H2>
<p>What do you know about Zeppelins?<BR>
</TD></TR><TR><TD><P>

<P><BR><H2><I>What is that?</I></H2>
<p>Lots of things, most of them interesting.<BR>
</TD></TR><TR><TD><P>

<P><BR><H2><I>Jumping Electrons!</I></H2>
<p>Getting&nbsp;from one place to another. . . my hotlists.<BR>
<P>
</TD></TR><TR>

<TD COLSPAN=4><P ALIGN=Center>
<P ALIGN=Center><BR>Send me
mail:<A HREF="MAILTO:mud@gonzo.wolfenet.com">mud@gonzo.wolfenet.com</A>
</TD></TR></TABLE></CENTER><P>
<P ALIGN=Center><BR>

<P>
</BODY></HTML>
```

Test Your Response Time

http://www.stack.urc.tue.nl/~jasperz/js/

Jasper van Zandbeek, jasperz@stack.urc.tue.nl

Changes background color after a random period of time.



Location: http://www.stack.urc.tue.nl/~jasperz/js/response.html

Test your Response time!

Test your Response time! Press start to start the test. Press stop when the background color changes. You can select the color in which the background changes. Try it! Test your response time! Press start and the background will change within 20 seconds!

Change background color in: crimson

Last modification of this page: **Mon Feb 26 09:55:36 1996**

Connection statistics are maintained by

```
<html>
<!-- author: Jasper van Zandbeek -->

<!-- Created: 25-2-1996-->

<!-- This document has been created by pub2html -->
<head><title>Test your Response time!</title>
<script language="JavaScript">
<!-- hiding for old browsers
       // response time test, created by Jasper van Zandbeek
       // e-mail: jasperz@stack.urc.tue.nl

var startTime=new Date();
var endTime=new Date();
var startPressed=false;
var bgChangeStarted=false;
var maxWait=20;
var timerID;

function startTest()
{

       document.bgColor=document.response.bgColorChange.options[document.response.
➥bgColorChange.selectedIndex].text;
       bgChangeStarted=true;
       startTime=new Date();
}
```

```
function remark(responseTime)
{
        var responseString="";
        if (responseTime < 0.10)
                responseString="Well done!";
        if (responseTime >= 0.10 && responseTime < 0.20)
                responseString="Nice!";
        if (responseTime >=0.20 && responseTime < 0.30)
                responseString="Could be better...";
        if (responseTime >=0.30 && responseTime < 0.60)
                responseString="Keep practicing!";
        if (responseTime >=0.60 && responseTime < 1)
                responseString="Have you been drinking?";
        if (responseTime >=1)
                responseString="Did you fall asleep?";

        return responseString;
}

function stopTest()
{
        if(bgChangeStarted)
        {
                endTime=new Date();
                var responseTime=(endTime.getTime()-startTime.getTime())/1000;

                document.bgColor="white";
                alert("Your response time is: " + responseTime + " seconds " + "\n"
➥+remark(responseTime));
                startPressed=false;
                bgChangeStarted=false;
        }
        else
        {
                if (!startPressed)
                {
                        alert("press start first to start test");
                }
                else
                {
                        clearTimeout(timerID);
                        startPressed=false;
                        alert("cheater! you pressed too early!");
                }
        }
}
```

```
var randMULTIPLIER=0x015a4e35;
var randINCREMENT=1;
var today=new Date();
var randSeed=today.getSeconds();
function randNumber()
{
      randSeed = (randMULTIPLIER * randSeed + randINCREMENT) & 0xffffffff;
      return((randSeed >> 16) & 0x7fff) / 32767;
}

function start()
{
      if(startPressed)
      {
            alert("Already started. Press stop to stop");
            return;
      }
      else
      {
            startPressed=true;
            timerID=setTimeout('startTest()', 20000*randNumber());
      }
}
// -->
</script>
</head><body bgcolor=#ffffff text=#000000 link=#0000ff vlink=#0000ff><center>
<table border=0>
      <tr>
            <td><a href="/~jasperz/index.html"><img src="/~jasperz/misc/b_home.gif"
➥alt="[home]" border=0 width=97 height=43></a></td>
            <td><a href="/~jasperz/js/index.html"><img src="/~jasperz/misc/b_js.gif"
➥alt="[javascript]" border=0 width=97 height=43></a></td>
            <td><a href="/~jasperz/js/response.html"><img src="/~jasperz/misc/
➥r_resp.gif" alt="[response]" border=0 width=97 height=43></a></td>
            <td><a href="/~jasperz/js/memory/index.html"><img src="/~jasperz/misc/
➥r_mem.gif" alt="[memory]" border=0 width=97 height=43></a></td>
            <td><a href="/~jasperz/js/puzzle/index.html"><img src="/~jasperz/misc/
➥r_puz.gif" alt="[puzzle]" border=0 width=97 height=43></a></td>
            <td><a href="/~jasperz/js/hanoi.html"><img src="/~jasperz/misc/
➥r_hanoi.gif" alt="[hanoi]" border=0 width=97 height=43></a></td>
            <td><a href="mailto:jasperz@stack.urc.tue.nl"><img src="/~jasperz/misc/
➥b_mail.gif" alt="[mail me]" border=0 width=97 height=43></a></td>
      </tr>
</table>
</center>
<p><center><h1>Test your Response time!</h1></center>

Test your Response time! Press start to start the test.
Press stop when the background color changes.
```

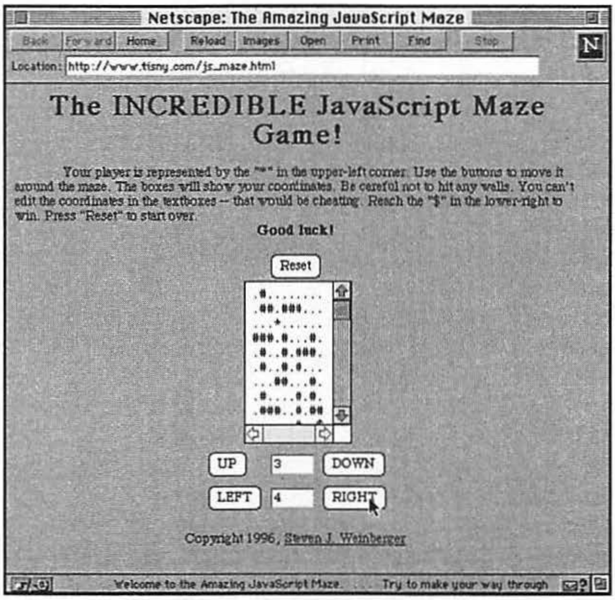
```
You can select the color in which the background changes. Try it! Test your response
➥time!
Press start and the background will change within 20 seconds!<p>

<form name="response">
Change background color in:
<select name="bgColorChange">
<option selected>deeppink
<option>aliceblue
<option>crimson
<option>darkkhaki
<option>cadetblue
<option>darkorchid
<option>coral
<option>chocolate
<option>mediumslateblue
<option>tomato
<option>darkslategray
<option>limegreen
<option>cornflowerblue
<option>darkolivegreen
</select>
<input type="button" value="start" onClick="start()">
<input type="button" value="stop" onClick="stopTest()">
</form>

<center><p>Last modification of this page: <b>Mon Feb 26 09:55:36 1996</b></center><p>
<center>Connection statistics are maintained by<br><a href="http://
➥www.stack.urc.tue.nl:81/~jasperz/cgi-bin/jc/jstat.cgi?name=js.response"><img
src="http://www.stack.urc.tue.nl:81/~jasperz/cgi-bin/jc/jc.cgi?name=js.response"
➥alt="JC" border=0></a>
</center><p>
</body></html>
```

The Amazing JavaScript Maze

http://www.tisny.com/js_maze.html

Steven J. Weinberger, weinberg@yu1.yu.edu

JavaScript is used to hold the data for every position in the maze, dynamically update the data based on user input, and fill the <TEXTAREA> with the actual maze.


Netscape: The Amazing JavaScript Maze
Back
Forward
Home
Reload
Images
Open
Print
Find
Stop
N
Location: http://www.tisny.com/js_maze.html
Reset
UP
DOWN
LEFT
RIGHT

The INCREDIBLE JavaScript Maze Game!

Your player is represented by the "*" in the upper-left corner. Use the buttons to move it around the maze. The boxes will show your coordinates. Be careful not to hit any walls. You can't edit the coordinates in the textboxes -- that would be cheating. Reach the "$" in the lower-right to win. Press "Reset" to start over.

Good luck!

```
.#........
.##.###...
...*......
###.#...#.
.#..#.###.
.#..#.#...
...##...#.
.#....#.#.
.###..#.##
```

3

4

Copyright 1996, Steven J. Weinberger

Welcome to the Amazing JavaScript Maze. . . . Try to make your way through

```
<html>
<head>
<title>The Amazing JavaScript Maze</title>

<!-- begin Script here

     Copyright 1996 by Steven Weinberger of Transaction Information
     Systems. All rights reserved.

     I can be reached at: steve@garcia.tisny.com, weinberg@yu1.yu.edu,
     vidkid@inch.com for comments, suggestions, etc.

     Free permission is given to re-use or modify this code,
     if and only if this copyright notice is kept in.

     This program/page illustrates the ability to create a grid on-screen,
     and plot and animate in it.  The main parts can be used to build a
     variety of useful programs (mostly games). -->

<script>
var line = "";
var x = 0;
var y = 0;
var full="*";
var blank = ".";
var wall = "#";
var goal = "$";
var fill = "";

// Functions to create the board

function makeboard() {
        for (var i=1; i<= 10; i++)
                this[i] = new makeRow();
        return this;
}

function makeRow() {
        for (var i=1; i<= 10; i++)
                this[i]=blank;
        return this;
}

// Functions to fill & clear the board.

function clearBoard (form) {
// Clears & resets the board
        x = 0;
        y = 0;
        form.xval.value = 1;
        form.yval.value = 1;
        for (var i=1; i<= 10; i++)
                for (var j=1; j<= 10; j++)
                        theBoard[i][j]=blank;
        drawMaze();
        fillBoard(form);
        return;
}

function fillBoard (form) {
        // Clear board buffer
        line = "";
        form.grid.value = "";
        // Fill board buffer
        for (var i=1; i<= 10; i++)
                for (var j=1; j<= 10; j++)
                        line += theBoard[i][j];
        // Move buffer contents to board
        form.grid.value=line;
}

function plot (v, h) {
         theBoard[v][h] = fill;
}
```

```
function drawMaze() {
// Plots the walls of the maze
//
// Ideally, a function should do this automatically,
// or maybe I should write a maze generating function in JS!
// Note: This program operates in Y,X co-ordinates (not standard X,Y).

        theBoard[10][10] = goal;
        theBoard[1][2] = wall;
        theBoard[2][2] = wall;
        theBoard[4][1] = wall;
        theBoard[4][2] = wall;
        theBoard[4][3] = wall;
        theBoard[2][3] = wall;
        theBoard[5][2] = wall;
        theBoard[6][2] = wall;
        theBoard[2][5] = wall;
        theBoard[4][5] = wall;
        theBoard[5][5] = wall;
        theBoard[2][6] = wall;
        theBoard[2][7] = wall;
        theBoard[9][10] = wall;
        theBoard[9][9] = wall;
        theBoard[8][9] = wall;
        theBoard[7][9] = wall;
        theBoard[10][7] = wall;
        theBoard[9][7] = wall;
        theBoard[8][7] = wall;
        theBoard[6][7] = wall;
        theBoard[9][2] = wall;
        theBoard[9][3] = wall;
        theBoard[9][4] = wall;
        theBoard[8][2] = wall;
        theBoard[7][4] = wall;
        theBoard[7][5] = wall;
        theBoard[6][5] = wall;
        theBoard[5][7] = wall;
        theBoard[5][8] = wall;
        theBoard[5][9] = wall;
        theBoard[4][9] = wall;
}

function update(form) {
        var horiz = eval(form.xval.value);
        var vert = eval(form.yval.value);
        plot(vert,horiz);
        fillBoard(form);
        return;
}
```

```
function initBoard() {
        theBoard = new makeboard();
        fill = full;
        clearBoard(document.board);
        update(document.board);
}

// Functions to handle the player piece
//
// I suppose I could have written one function to handle this,
// but it was getting too complex.  Feel free to try. :)
//

function decx(form) {
        fill = blank;
        update(form);
        checkx = eval(form.xval.value - 1);
        checky = form.yval.value;
        if (form.xval.value > 1) {
                if (theBoard[checky][checkx] != wall) {
                        form.xval.value=eval(form.xval.value - 1);
                }
                else {
                        alert("THUD!\nYou hit a wall.");
                }
                if (theBoard[checky][checkx] == goal) {
                        alert("YOU WIN!");
                         location.href="http://www.tisny.com/js_demo.html";
                }
        }
        fill = full;
        update(form);
}

function incx(form) {
        fill = blank;
        update(form);
        checkx = eval(1 * form.xval.value + 1);
        checky = form.yval.value;
        if (form.xval.value < 10) {
                if (theBoard[checky][checkx] != wall) {
                        form.xval.value=eval(1 * form.xval.value + 1);
                }
                else {
                        alert("THUD!\nYou hit a wall.");
                }
                if (theBoard[checky][checkx] == goal) {
                        alert("YOU WIN!");
                         location.href="http://www.tisny.com/js_demo.html";
                }
        }
        fill = full;
        update(form);
}

function decy(form) {
        fill = blank;
        update(form);
        checkx = form.xval.value;
        checky = eval(form.yval.value - 1);
        if (form.yval.value > 1) {
                if (theBoard[checky][checkx] != wall) {
                        form.yval.value=eval(form.yval.value - 1);
                }
                else {
                        alert("THUD!\nYou hit a wall.");
                }
                if (theBoard[checky][checkx] == goal) {
                        alert("YOU WIN!");
                         location.href="http://www.tisny.com/js_demo.html";
                }
        }
        fill = full;
        update(form);
}

function incy(form) {
        fill = blank;
        update(form);
        checkx = form.xval.value;
        checky = eval(1 * form.yval.value + 1);
        if (form.yval.value < 10) {
                if (theBoard[checky][checkx] != wall) {
                        form.yval.value=eval(1 * form.yval.value + 1);
                }
                else {
                        alert("THUD!\nYou hit a wall.");
                }
                if (theBoard[checky][checkx] == goal) {
                        alert("YOU WIN!");
                         location.href="http://www.mcp.com/hayden/";
                }
        }
        fill = full;
        update(form);
}
```

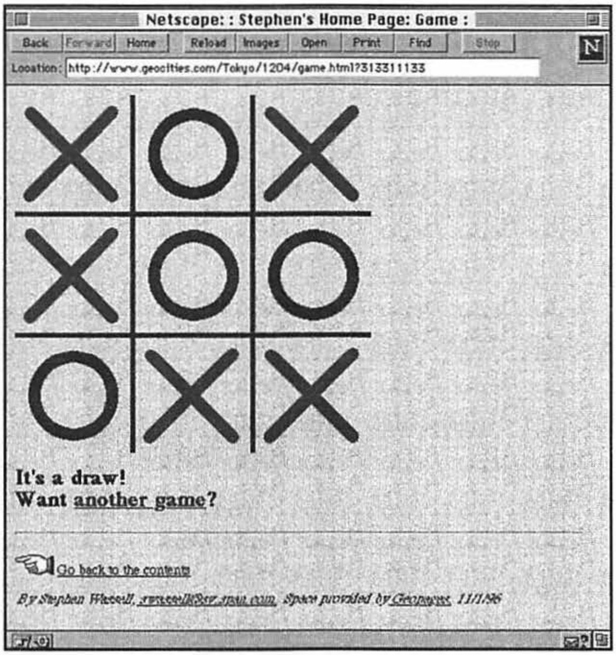
```
// Various Functions

function cheater (form) {
// Refuse to change values manually, and start over. CHEATER!
        alert("You can't change this value manually.\nPlease use the buttons.");
        clearBoard(form);
        update(form);
}

// Scrolling Status Bar
// This scrolling status bar was taken from public domain
// I make no claims on it, and placed it here as an enhancement to my page
// My apologies to the author, for forgetting to keep his disclaimer.

function scrollit_r21(seed)
{
        var m1  = "Welcome to the Amazing JavaScript Maze.   . . .    ";
        var m2  = "Try to make your way through the Maze.   . . .    ";
        var m3  = "Don't hit any walls.   . . .    ";
        var m4  = "Good Luck   . . . ";

        var msg=m1+m2+m3+m4;
        var out = " ";
        var c   = 1;

        if (seed > 100) {
                seed--;
                 var cmd="scrollit_r21(" + seed + ")";
                 timerTwo=window.setTimeout(cmd,100);
        }
        else if (seed <= 100 && seed > 0) {
                for (c=0 ; c < seed ; c++) {
                        out+=" ";
                }
                out+=msg;
                seed--;
                 var cmd="scrollit_r21(" + seed + ")";
                     window.status=out;
                 timerTwo=window.setTimeout(cmd,100);
        }
        else if (seed <= 0) {
                if (-seed < msg.length) {
                         out+=msg.substring(-seed,msg.length);
                        seed--   ;
                         var cmd="scrollit_r21(" + seed + ")";
                        window.status=out;
                         timerTwo=window.setTimeout(cmd,100);
                }
                else {
                        window.status=" ";
                         timerTwo=window.setTimeout("scrollit_r21(100)",75);
                }
        }
}

// End of functions
</script>
<body onLoad="timerONE=window.setTimeout('scrollit_r21(100)',500);initBoard();">
<center>
<h1>The INCREDIBLE JavaScript Maze Game!</h1>
</center>
<dd>Your player is represented by the "*" in the upper-left corner.
Use the buttons to move it around the maze.  The boxes will show your
coordinates. Be careful not to hit any walls. You can't edit the coordinates
in the textboxes -- that would be cheating. Reach the "$" in the lower-right
to win.
Press "Reset" to start over.<br>
<center>
<b>Good luck!</b>
<p>
<form method="post" name="board">
<input type='button' value='Reset'
onClick='clearBoard(this.form);update(document.board);'>
<br>
<textarea name="grid" rows="9" cols="10" wrap=virtual></textarea><br>
<!-- virtual-wrap is the key! Now one text line becomes a grid! -->
<table>
<tr>
  <td><input type='button' value='UP' onClick='decy(this.form)'></td>
  <td><input type='text' value='1' size=5 name='yval'onChange='cheater(this.form);
➥'></td>
  <td><input type='button' value='DOWN' onClick='incy(this.form)'></td>
<tr>
  <td><input type='button' value='LEFT' onClick='decx(this.form)'></td>
  <td><input type='text' value='1' size=5 name='xval'
onChange='cheater(this.form);'></td>
  <td><input type='button' value='RIGHT' onClick='incx(this.form)'></td>
</table>
</form>
<p>
Copyright 1996, <a href="mailto:weinberg@yu1.yu.edu">Steven J. Weinberger</a>
</center>
</body>
</html>
```

JavaScript Noughts & Crosses

http://www.geocities.com/Tokyo/1204/game.html

Stephen Wassell, swassell@sv.span.com

Tic-Tac-Toe on the Web! JavaScript works out the next move, generates the board, and creates links for each blank to reload the page with new data.



Location: http://www.geocities.com/Tokyo/1204/game.html?313311133

It's a draw!
Want another game?

Go back to the contents

By Stephen Wassell, swassell@sv.span.com, Space provided by Geopages, 11/1/96

```
<html>
<head>
<title>: Stephen's Home Page: Game :</title>
</head>

<body bgcolor ="#E0FFC0" background="base.gif">

<script language="LiveScript"> <!--

//Written by Stephen Wassell
//swassell@sv.span.com
//http://www.geocities.com/Tokyo/1024
//A JavaScript version of Noughts and Crosses
```

```
//Tested on Netscape 2.0b4

//location.search holds the board in the format '?111111111'
//each number is a square on the board:
//0 = nought (computer), 1 = blank, 2 = cross (human)
//the program uses the sum of these to work out where to go
//it doesn't bother checking for human wins as it can't lose
//I bet Eric won't believe me, though :)

//can this language really have no arrays?
//I've had to use substrings instead - not good :-)

function Get (Str, Off) { //equiv. to return Str[Off]
//'- -' turns a string into a number
    return - -Str.substring (Off, Off+1)
}

function Set (Str, Off, Val) { //equiv. to Str[Off] = Val; return Str
    return Str.substring (0, Off) + Val + Str.substring (Off+1, 10)
}

function Sum (Str, a, b, c) { //adds the contents of a, b and c
    return (Get (Str, a) + Get (Str, b) + Get (Str, c))
}

function MyMove (Dat) { //do computer's move
    var PosLines, Order = '2613', PosCorns = '124326748968'
    var j, i, a, b, c

        if (Get (Dat, 5) == 1) //if computer's in centre
               PosLines = '132798174396546528519537'
        else
               PosLines = '519537132798174396546528'

    Result = ResWin

    for (j = 0; j < 4; j++) {
        for (i = 0; i < 24; i += 3) {

            a = Get (PosLines, i)
            b = Get (PosLines, i + 1)
            c = Get (PosLines, i + 2)
            if (Sum (Dat, a, b, c) == Get (Order, j)) {
                         if (Get (Dat, a) == 0) return Set (Dat, a, 1)
                         if (Get (Dat, b) == 0) return Set (Dat, b, 1)
                         if (Get (Dat, c) == 0) return Set (Dat, c, 1)
            }
        }
            Result = ResNorm
```

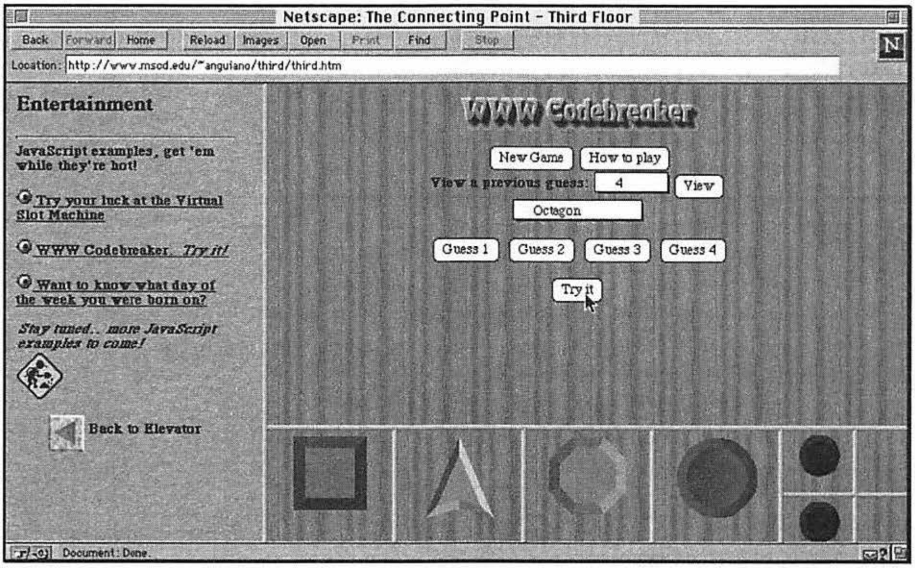
```
           if (j == 1) { //only between 2nd and 3rd passes
                              for (i = 0; i < 12; i += 3) {
                                     a = Get (PosCorns, i)
                                     b = Get (PosCorns, i + 1)
                                     c = Get (PosCorns, i + 2)
                                     if (Sum (Dat, a, b, c) == 6)
                                     if (Get (Dat, a) == 0) return Set (Dat, a, 1)
                                }
                          }
    }

    Result = ResDraw //no places to go
    return Dat
}

function DrawTable (Dat) { //plots the grid
    var i, Sqr

    for (i = 1; i <= 9; i++) {
        Sqr = Get (Dat, i)

        if (Sqr == 0) { //it's a blank
            if (Result == ResWin) //no more links if it's been won
                document.write ('<IMG ALIGN=bottom SRC="gameb.gif">')
            else {
                document.write ('<A HREF="game.html') //a link for human moves
                document.write (Set (Dat, i, '3'))
                 document.write ('"><IMG ALIGN=bottom BORDER=0 SRC="gameb.gif"></A>')
            }

        } else if (Sqr == 3) //it's an X
             document.write ('<IMG ALIGN=bottom SRC="gamex.gif">')
        else //must be an O
             document.write ('<IMG ALIGN=bottom SRC="gameo.gif">')

        if (i == 9) //finished
            document.write ('<P>')
        else if (i == 3 || i == 6) //long horizontal line
             document.write ('<BR><IMG ALIGN=bottom SRC="gamedh.gif"><BR>')
        else //vertical line
             document.write ('<IMG ALIGN=bottom SRC="gamedv.gif">')
    }

    document.write ('<H2>') //make comments

    if (Result == ResDraw)
        document.write ("It's" + ' a draw!<BR>Want <A HREF="game.html">another game
        ➥</A>?')
    else if (Result == ResStart)
        document.write ("Your go first...")

    else if (Result == ResWin)
        document.write ('I won!<BR>Like to <A HREF="game.html">play again</A>?')

    document.write ('<BR></H2>')
}

var ResNorm = 0, ResWin = 1, ResDraw = 2, ResStart = 3
var Result = ResStart

if (location.search.length == 10) //during game
    DrawTable (MyMove (location.search))
else //just started
    DrawTable ('?000000000') //draw a load of blanks

// --> <H2>Sorry! You need a <A HREF="http://home.netscape.com/comprod/products/
➥navigator/version_2.0/">JavaScript-capable browser</A> to run this.</H2>

</script>

<P>
<HR>
<P>
<IMG ALIGN=abscentre SRC="back.gif"> <A HREF="index.html">Go back to the contents</A>
<P>
<I>By Stephen Wassell, <A HREF = "mailto:swassell@sv.span.com">swassell@sv.span.com
➥</A>. Space provided by <A HREF="http://www.geopages.com/">Geopages</A>. 11/1/96</I>

</body>
</html>
```

The Connecting Point: WWW Codebreaker

http://www.mscd.edu/~anguiano/third/third.htm

Jason Anguiano, jangui@csn.net

Receiving input from users, displaying different graphics files in different frames, scrolling text.


Netscape: The Connecting Point - Third Floor

Location: http://www.msod.edu/~anguiano/third/third.htm

Entertainment

JavaScript examples, get 'em while they're hot!

Try your luck at the Virtual Slot Machine

WWW Codebreaker. *Try it!*

Want to know what day of the week you were born on?

Stay tuned.. more JavaScript examples to come!

Back to Elevator

View a previous guess: 4

Octagon

Document: Done.

```
<HTML>
<HEAD>
<TITLE>The Connecting Point - WWW Codebreaker</TITLE>
</HEAD>
<FRAMESET ROWS="75%,25%">
<FRAME SRC="code.htm" NAME="Main_Code" NORESIZE>
<FRAMESET COLS="20%,20%,20%,20%,20%">
<FRAME SRC="square3d.htm" NAME="Guess1" NORESIZE SCROLLING="no" MARGINWIDTH="5"
➥MARGINHEIGHT="5">
<FRAME SRC="cir3d.htm" NAME="Guess2" NORESIZE SCROLLING="no" MARGINWIDTH="5"
➥MARGINHEIGHT="5">
<FRAME SRC="arrow3d.htm" NAME="Guess3" NORESIZE SCROLLING="no" MARGINWIDTH="5"
➥MARGINHEIGHT="5">
<FRAME SRC="octgn3d.htm" NAME="Guess4" NORESIZE SCROLLING="no" MARGINWIDTH="5"
➥MARGINHEIGHT="5">

<FRAMESET COLS=35%,65%">
       <FRAMESET ROWS=60%,40%">
              <FRAME SRC="slotempt.htm" NAME="Answer1" NORESIZE SCROLLING="no"
➥MARGINWIDTH="1" MARGINHEIGHT="1">
              <FRAME SRC="slotempt.htm" NAME="Answer2" NORESIZE SCROLLING="no"
➥MARGINWIDTH="1" MARGINHEIGHT="1">
        </FRAMESET>
        <FRAMESET ROWS=60%,40%">
               <FRAME SRC="slotempt.htm" NAME="Answer3" NORESIZE SCROLLING="no"
➥MARGINWIDTH="1" MARGINHEIGHT="1">
               <FRAME SRC="slotempt.htm" NAME="Answer4" NORESIZE SCROLLING="no"
➥MARGINWIDTH="1" MARGINHEIGHT="1">
```

```
        </FRAMESET>
</FRAMESET>
</FRAMESET>
</FRAMESET>

</HTML>
```

code.htm

```
<HTML>
<HEAD>
<BODY BACKGROUND="wood.gif" TEXT="#000000" LINK="#0000F6" VLINK="#006B03"
➥ALINK="#EE0000">
<TITLE>The Connecting Point - WWW Codebreaker</TITLE>

<SCRIPT LANGUAGE="LiveScript">
<!--
//Author:  Jason Anguiano
//Date:    February 25, 1996
//Email:   jangui@csn.net
//Program: Codebreaker for the World Wide Web. Written in JavaScript

/* This JavaScript can be re-used or modified, if credit is given in
   the source code. Thank you.

   I cannot be held responsible for any unwanted effects due to the
   usage of this JavaScript or any derivative.  No warrantees for usability
   for any specific application are given or implied.

   Sorry about that, ...now where were we?
*/
//
//Declarations
//=====================================================================
   var play = 1;
   var win =  0;
   var view = 2;
   var lose = 3;
   var game_state = win;  //Initially so that the player cannot play until
                     //a new game is started
//- - - - - - - - - - -
   var empty = 6;
   var black = 7;
   var white = 8;
//- - - - - - - - - - -
   var guess_num;       //Keeps track of the current guess
//- - - - - - - - - - -
```

```
   //guess is the base for a two dimensional array.
   //Each item in the guess array is actually another array of four items.
   //The number 10 is used because the player gets only 10 guesses.
   guess= new MakeArray(10);
   answer= new MakeArray(4);
//========================================================================

//Create a random number between 0 and 1
function RandomNumber0_1() {
  today = new Date();   //Seed
  myseed = today.getTime() % 1000; //cut the number down in size

  return Math.abs(Math.sin(myseed));
}

//=========================================================================
//Create a random number
function RandomNumber() {
  today = new Date();   //Seed
  return today.getTime() % 1000; //cut the number down in size
}

//=========================================================================
//This function will create an array of size n
function MakeArray(n) {
   this.length = n;
   for (var i = 1; i <= n; i++) {
     this[i] = 0 }
   return this
}

//=========================================================================
//display_item displays an image file in a certain frame based on two
//variables. loc is the location of the frame, and x is the image to
//display
function display_item(loc, x){
   if (x == 0)
     parent.frames[loc].location.href="square3d.htm";
   else if (x == 1)
     parent.frames[loc].location.href="cir3d.htm";
   else if (x == 2)
     parent.frames[loc].location.href="arrow3d.htm";
   else if (x == 3)
     parent.frames[loc].location.href="pent3d.htm";
   else if (x == 4)
     parent.frames[loc].location.href="hex3d.htm";
   else if (x == 5)
     parent.frames[loc].location.href="octgn3d.htm";
   else if (x == empty)
     parent.frames[loc].location.href="slotempt.htm";
   else if (x == black)
     parent.frames[loc].location.href="blckpeg.htm";
   else if (x == white)
     parent.frames[loc].location.href="whtpeg.htm";
   return true;
}

//=========================================================================
//calculate_pegs evaluates the guess against the actual answer and
//determines the number of black, white, or no pegs to be displayed.
function calculate_pegs(mynum){
   black_num = 0;
   white_num = 0;

   //A temporary array is created to hold the anwser. This allows the
   //answer to be modified during calculation without destroying the
   //actual answer.
   temp_answer = new MakeArray(4);
   for(var i=1; i<=4; i++)
      temp_answer[i] = answer[i];

   //How many white pegs?
   for(i=1; i<=4; i++){
      for(var j=1; j<=4; j++){
         if (guess[mynum][i] == temp_answer[j]){
            white_num++;
            temp_answer[j] = -1;
            break;
         }
      }
   }

   //How many black pegs?
   for(i=1; i<=4; i++){
      if (guess[mynum][i] == answer[i]){
         white_num--;
         black_num++;
      }
   }

   //Display the black and white answer pegs
   var temp_black = black_num;
   var temp_white = white_num;
   for(i=5; i<=8; i++){
      if (temp_black>0){
         display_item(i, black);
         temp_black--;
      }
      else if (temp_white>0){
         display_item(i, white);
         temp_white--;
      }
      else
         display_item(i, empty);
   }

   //Change the state of the game to win if there are 4 black pegs.
   if (black_num == 4){
      game_state = win;
   }
   return true;
}

//========================================================================
//Show the appropriate four graphic files according to the entire guess
//by the player. mynum represents which guess is to be displayed.
function display_guess(mynum){
   //display the current guess
   for(var i = 1; i <=4; i++){
      display_item(i, guess[mynum][i]);
   }
   //Recalculate the pegs
   if (mynum != guess_num){
      calculate_pegs(mynum);
   }
   else{
      calculate_pegs(mynum - 1);
      alert("These answer pegs represent the previous guess");
   }
   return true;
}

//========================================================================
//Using drop-down boxes is probably not the best way to select a guess to
//view. I did it this way as an example of how to use drop-down boxes and
//the selectedIndex property.
function change_view(form){
 form.Shape.select
 if ((game_state == play)¦¦(game_state == view)){

  //Cant select beyond the current guess...
  if((form.MyGuess.selectedIndex) < (guess_num-1)){
    var guess_view = form.MyGuess.selectedIndex;
    guess_view++;
    display_guess(guess_view);
    game_state = view;
  }
  //...If they do, then set the view back to the current guess
  else{
    form.MyGuess.options[guess_num-1].selected = true;
    display_guess(guess_num);
    game_state = play;
  }
 }
}

//========================================================================
//Randomly select an answer
function make_answer(my_array){
   var i;
   for(i=1; i<=4; i++){
        my_array[i] = parseInt(RandomNumber()%7); //seven possible choices

        //now timeout for a random period of time before selecting
        //another answer
        var mycount = parseInt(RandomNumber0_1()*1000);
        for(var j = 0; j < mycount; j++);
   }
   return true;
}

//========================================================================
//The player has guessed a shape for a certain location. Display that shape
//in location i.
function guess_shape(form,i){
 if (game_state == play){
   guess[guess_num][i] = form.Shape.selectedIndex;
   display_item(i, guess[guess_num][i]);
   return true;
 }
 return false;
}

//========================================================================
//The player submitted a four code combination guess. Now, analyze the
//guess and give the player feedback.
function guess_answer(form){
 if (game_state == play){
   //Calculate the black and white pegs
   calculate_pegs(guess_num);

   //Did the player win?
   if (game_state == win){
      alert("You win!");
      return true;
   }

   //The player did not win...
   else{
      //Increment the guess number
      guess_num++;

      //Did the player exceed the maximum number of guesses?
      if (guess_num > 10){
         alert("Sorry, Game Over.. Try Again! Here is the correct combination:");
         game_state = lose;
         //Show the winning combination
         for(var i =1; i <=4; i++){
            display_item(i, answer[i]);
         }
         return true;
      }

      //The game continues...
      //copy contents of previous guess into next guess
      for(i=1; i<=4; i++){
         guess[guess_num][i] = guess[guess_num - 1][i];
      }
      form.MyGuess.options[guess_num-1].selected = true;
      return true;
   }
 }
 return false;
}

//====================================================================
function new_game(form){
   game_state = play;
   guess_num = 1;

   //Select the first guess from the drop down box
   form.MyGuess.options[guess_num-1].selected = true;
   //Create the two-dimensional guess array
   for(var i =1; i <=10; i++){
      guess[i] = new MakeArray(4);
      for(var j =1; j<=4; j++){
         guess[i][j] = empty;
      }
   }

   //Initialize the answer array
   for(i=1; i<=4; i++){
      answer[i] = empty;
   }

   //Clean the board
   for(i=1; i<=8; i++){
      display_item(i, empty);
   }

   //randomly pick an answer
   make_answer(answer);

   return true;
}

//========================================================================
//Display the help file
function showhelp(form){
   parent.frames[0].location.href="codehelp.htm"
}
//========================================================================
//-->
</SCRIPT>

</HEAD>
<BODY>
<CENTER><IMG SRC="code.gif" WIDTH=201 HEIGHT=30></CENTER>

<FORM NAME = "WWWCB">
<CENTER>

<INPUT TYPE="button" name="NewGame" value="New Game" onclick="new_game(this.form)">
<INPUT TYPE="button" name="Help1" value="How to play"
➥onclick="showhelp(this.form)"><BR>
<B>View a previous guess: </B><SELECT NAME="MyGuess">
<OPTION> 1
<OPTION> 2
<OPTION> 3
```

```
<OPTION> 4
<OPTION> 5
<OPTION> 6
<OPTION> 7
<OPTION> 8
<OPTION> 9
<OPTION> 10
</SELECT>
<INPUT TYPE="button" name="MyView" value="View" onclick="change_view(this.form)">
<BR>
<SELECT NAME="Shape">
<OPTION> Square
<OPTION> Circle
<OPTION> Arrow
<OPTION> Pentagon
<OPTION> Hexagon
<OPTION> Octagon
<OPTION> Empty
</SELECT>
<BR>
<PRE>
<INPUT TYPE="button" name="Guess1" value="Guess 1" onclick="guess_shape(this.form,1)">
➥<INPUT TYPE="button" name="Guess2" value="Guess 2" onclick="guess_shape(this.form,2)">
➥<INPUT TYPE="button" name="Guess3" value="Guess 3" onclick="guess_shape(this.form,3)">
➥<INPUT TYPE="button" name="Guess4"  value="Guess 4" onclick="guess_shape(this.form,4)">
</PRE>
<INPUT TYPE="button" name="Guess" value="Try it" onclick="guess_answer(this.form)">
</CENTER>
</FORM>

</BODY>
</HTML>
```

Rainbow Text

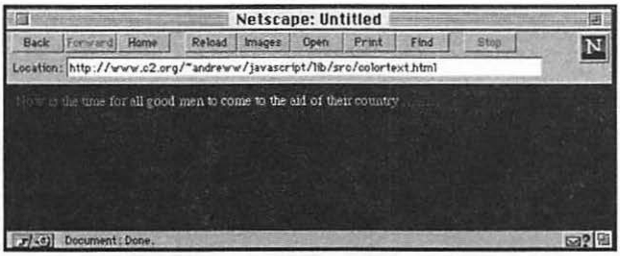
http://www.c2.org/~andreww/javascript/lib/src/colortext.html (small example)

http://www.tfh-berlin.de/~maze/index.html (part of larger Web page)

Mathias Hoeschen, maze@tfh-berlin.de

The `FadeText()` function makes the text look rainbow-colored. This is done by fading the R, G, and B parts of the text color in and out with some math functions.

```
<HTML>
```



Location: http://www.c2.org/~andreww/javascript/lib/src/colortext.html

Now is the time for all good men to come to the aid of their country ......

```
<HEAD>
<TITLE>Untitled</TITLE>

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

 // Feel free to modify or copy this Script for non-commercial use, but please leave
➥this comment in the code.
// To use the Fade()-function in HTML-code, always include the HTML-comment-tags to
➥the output-string!!!
// This is not necessary for Browsers understanding Java, but to display the text in
➥other browsers too, they are needed!
// Nice trick, eh?
//
// written and  by (c) Mathias Hoeschen, Tel./FAX: +49 30 6283675, maze@tfh-berlin.de,
Home: http://www.tfh-berlin.de/~maze/
// on 20.Feb.1996

function MakeArray(n){
   this.length=n;
   for(var i=1; i<=n; i++) this[i]=i-1;
   return this
}

hex=new MakeArray(16);
hex[11]="A"; hex[12]="B"; hex[13]="C"; hex[14]="D"; hex[15]="E"; hex[16]="F";

function ToHex(x){          // Converts a int to hex (in the range 0...255)
   var high=x/16;
   var s=high+"";           //1.
   s=s.substring(0,2);             //2. These three lines do the same as a 'trunc'-
function (because there is no trunc, unfortunately!)
   high=parseInt(s,10);            //3.
   var left=hex[high+1];    // left part of the hex-value
   var low=x-high*16;       // calculate the rest
   s=low+"";                //1.
   s=s.substring(0,2);             //2. see above
   low=parseInt(s,10);             //3.
   var right=hex[low+1];    // right part of the hex-value
   var string=left+""+right;     // put the high and low together
   return string;
}

function Fade(text){
   text=text.substring(3,text.length-4); // removes the HTML-comment-tags
   color_d1=255;                         // can be any value in 'begin'...255
   mul=color_d1/text.length;
   for(i=0;i<text.length;i++){
      color_d1=255*Math.sin(i/(text.length/3));   // "=255-mul*i" to fade out,
➥"=mul*i" to fade in, or try "255*Math.sin(i/(text.length/3))"
      color_h1=ToHex(color_d1);
      color_d2=mul*i;
      color_h2=ToHex(color_d2);
      document.write("<FONT
➥COLOR='#FF"+color_h1+color_h2+"'>"+text.substring(i,i+1)+'</FONT>');
   }
}

//--></SCRIPT></HEAD>
<body bgcolor=black>
<SCRIPT LANGUAGE="JavaScript"><!--

Fade("-->Now is the time for all good men to come to the aid of their country
➥........<!--");

//--></SCRIPT><BR>

</BODY>
</HTML>
```

INDEX

About the CD-ROM

The *JavaScript for Macintosh* CD-ROM contains everything you need to begin your journey with JavaScript:

□ All the JavaScripts used in the book so that you can play around with them yourself

□ The complete code for the scripts shown in Appendix B, "JavaScripts from Around the Web"

□ BBEdit Lite—the premier freeware text editor

□ BBEdit 3.5.2 Demo

□ GifBuilder—freeware utility for creating GIF animations

Hayden Books

JavaScript— the next step for Web publishers

This is your premier guide to JavaScript, the hot new Web programming language that enables you to create interactive Web pages without the complications of Java or CGI. This tutorial takes you step-by-step through programming cross-platform JavaScripts to make your Web pages more dynamic, flexible, and interactive than ever before.

Detailed information on every aspect of the JavaScript language—from declarations and operations to document and history objects—will soon have you exploiting the ease and power of this dynamic, object-based language.

Detailed coverage of...

- Declarations and operations
- Functions and methods
- Flow control
- Document, history, window, and frame objects

THE COMPANION CD-ROM INCLUDES:

- COOL JAVASCRIPT SCRIPTS
- TUTORIALS FROM THE BOOK
- BBEDIT LITE AND BBEDIT 3.5.2 DEMO
- GIFBUILDER
- JAVASCRIPT EXAMPLES FROM AROUND THE WEB

Matt Shobe is a Macintosh® expert and has done Web site design, usability testing, and documentation work for SPRY/CompuServe®, Microsoft®, and Andersen Consulting LLP.

Tim Ritchey has worked on artificial intelligence, high-performance parallel architectures, and computer vision. His present interests include distributed computing, VRML, and of course Java.

$40.00 USA / $54.95 CAN / £37.50 NET UK (INC OF VAT)

ISBN
9 781568 302782 0 52059 02789 7

Category: Macintosh
Communications/Online—Internet
User Level: Intermediate—Advanced